NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Dissertation Title: Exploring Instruction Level Parallelism Using Resource Flow Execution
Author: David A. Morano
Department: Electrical and Computer Engineering

Approved for Dissertation Requirement of the Doctor of Philosophy Degree:

Dissertation Advisor: Professor David Kaeli	Date
Dissertation Committee: Professor Xinping Zhu	Date
Dissertation Committee: Professor Waleed Meleis	Date
Department Chair:	Date

Graduate School Notified of Acceptance:

Director of the Graduate School	Date

Copy Deposited in Library:

Reference Librarian	Date

EXPLORING INSTRUCTION LEVEL PARALLELISM USING RESOURCE FLOW EXECUTION

A Dissertation Presented
by

David A. Morano

to
The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in the field of
Electrical Engineering

Northeastern University
Boston, Massachusetts

May 2007

Abstract

Due to clock frequency limitations, performing more computer operations in parallel is now being more widely explored for increasing total execution performance. Several varieties of parallelism are available, with the easiest and most attractive being thread-level parallelism. Recently introduced microprocessors have provided capabilities such as: multiple threads multiplexed onto a single CPU (simultaneous multithreading or hyperthreading), multiple CPU cores on a single integrated circuit, multiple microprocessor integrated circuits placed on the same logic module or logic printed circuit board, or sometimes various combinations of all of these. Although these approaches increase the total computing throughput available, they do nothing for more serially dependent program workloads. Rather these approaches require workloads that can be highly or totally parallelizable.

There are machine design approaches that attempt to address serial workloads through the extraction of instruction level parallelism. One such approach is that of Explicit Parallel Instruction Computing (or EPIC for short). This approach attempts to execute several instructions in parallel through the use of clever scheduling by its compiler. However, substantial performance increases using this architectural technique have not been achieved to date. Further, these approaches require entirely new machine architectures and are therefore not appropriate for legacy programs and architectures.

In the present work, we present a way to both speed up those programs that are highly serial in nature (not generally parallelizable) and to be able to execute these programs on legacy machine architectures (not requiring a new architecture). We introduce a new machine microarchitectural framework that performs dynamic instruction scheduling in a new and more flexible way as compared with past superscalar machines. We term our new machine execution philosophy Resource Flow Execution. In this philosophy, instruction execution is performed in a very relaxed way and is not initially constrained by either control or data flow dependencies. Rather, we introduce novel machine microarchitectural components that allow for the dynamic determination of program dependencies during or after instruction execution. As dependencies are discovered, instructions get re-executed as necessary (and entirely in parallel with other instructions) so as to eventually converge on the expected committed architectural state of the program.

Acknowledgements

I would firstly like to thank Professor Augustus Uht of the University of Rhode Island. The basic foundation of this work as well as most of the ideas used to implement it were first introduced by him. More specifically he is responsible for the original idea of the Resource Flow execution model. He is responsible for extending the idea of the classic reservation station into the generalized instruction issue station, as well for the basic idea of allowing for rampart out-of-order execution that only converges to the committed program state through repeated execution of in flight instructions (all coordinated through the issue station component). In effect, it is he who elevated the operand to the primary focus that it has in the Resource Flow execution model. He also introduced the idea of using integer time-tags for dynamic dependency enforcement during execution. He also introduced the idea of generalized microarchitectural predication for all instructions in flight within the machine. Professor Uht is also largely responsible for the introduction of the Levo microarchitecture (presented within) with many of its current forms. Additionally, I would like to extend many thanks to Alireza Khalafi for his many contributions to many of the individual ideas within the larger Resource Flow framework. Specifically, he first advocated the elimination of a centralized architected register file with the adoption of the existing rotating provisional register file. Additionally, Alireza Khalafi is also responsible for recognizing the need for what became known as nullify-forwarding for memory operands. He also helped substantially on many other detailed microarchitectural decisions as well as with the advancement of the simulation frameworks used to carry out the present work. I would also like to thank Marcos de Alba for this early contribution with the design and the development of our more detailed simulation framework. I would also like to extend many thanks to my dissertation advisor, Professor David Kaeli, for his many contributions to the design and development of the Resource Flow execution model as well as for his contribution to the idea of dynamic microarchitectural predication. Additionally he provided many useful suggestions and help while developing several new machine components as well as with the general operation of the introduced machine microarchitectures (both OpTiFlow and Levo) as a whole. Finally I would like to thank all the members of my dissertation committee: Professors Walled Meleis, Zinping Zhu, and David Kaeli.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of computer designers from nearly the outset of the introduction of electronic computing machines has been to increase program execution performance. The importance of this goal has only intensified since then. For our purposes, execution performance is defined as the amount of computational work that can be computed (completed) per unit time. This ratio is also often termed the *speed* of the execution. The computational work to be completed is generally defined in a very high level and abstract way. A simple example of some work to be performed would be to find the solution to an abstract problem that can be computationally solved through the application of a computing algorithm.

In the early years of electronic computing, most research was still oriented towards what model programmable computing should take and what sorts of problems could be applied to automatic computing machines. However, after the introduction of both the stored program computer (which remains to the present day) and the fully electronic solid state computer, the industry started to focus more on advancing the speed of the executed programs rather than simply providing the means to represent am application problem with a program on a computing machine. Since then, the thirst for program execution speed has never been satisfied and, to the contrary, has rather dramatically increased with no foreseeable end in sight.

The first major upset in computing performance came with the introduction of the 6400, 6500, and 6600 series computers by Control Data Corporation (CDC) in 1963. There had been a consolidation of computing companies which left IBM largely in control of the entire market. IBM was in a position to be fairly confident that few other computer manufacturers could compete with it for general purpose business oriented machines. However, the introduction of the CDC 6600 caused a spirited disruption in the expectations for program execution performance. The CDC 6600 was able to substantially outperform the present offerings by IBM and the subsequent race

for speed by all competing computer machine vendors ensued. Coincidentally, the CDC 6600 especially also introduced a number of concepts in program execution and machine parallelism that have sparked thinking in machine design ever since. We are delighted to participate in the race for more computing speed and pursue, with this dissertation, the same avenue of attack for speed that was successfully used starting with the CDC 6600.

Our present goal is also to enhance program performance (computational work per unit time). Our basic approach is through the application of parallel execution of the program within the hardware of the machine itself. In the following sections we orient the reader to the specifics of our approach towards increasing program execution performance.

1.1 Orientation to execution performance

We start by separating out the components of the computing system that go into the final execution performance of a programmed application problem. The application problem may be one in which it may be very easily and naturally broken up into many parallel problems (subproblems) that have little or nothing to do with each other (little or no mutual dependencies between subproblems), other than perhaps proceeding from a common starting point consisting of input data of one sort or another. These problems are partitioned into several individual problems at the software level of the computing abstraction. These types of problems have come to be termed *embarrassingly parallel*. But these are not only relatively easy to solve using multiple existing computing machines (employed in parallel), they are also not representative of most complex problems facing the world today. For this reason we focus on the more difficult problems that have no easy means of being separated into separate subproblems for the purposes of computing a total result in parallel. These problems generally have complex control and data dependencies in their programs that greatly limit how they might be executed with anything other than a strictly sequential execution model. These are the class of application problems that we attempt to improve the execution performance of with our work. More specifically, they are single threaded programs with a relatively high degree of dependencies throughout most of their execution phases. These types of programs have variously been termed *sequential*, *integer*, or *integer sequential* programs.

When a single (and single-threaded) sequential program is mapped onto a computing machine, we can examine subcomponents that contribute to the execution performance of the machine. Again, execution performance is represented by the amount of computational work executed on the machine per unit time. This quantity can be further factored according to Equation 1.1.

$$\frac{work}{time} = \frac{work}{instr} \times \frac{instrs}{clock} \times \frac{clocks}{time} \tag{1.1}$$

Each of the component factors of execution performance (also termed the *speed* of the machine) is

roughly associated with a different aspect of the computing machine the program is executed on.

The amount of computational work performed per instruction (the first component factor $work/instr$) is largely a function of the architecture of the machine, and more specifically and more narrowly the instruction set architecture (ISA) of the machine. This is evident since the machine instructions can be designed in ways to perform more or less computational work for each instruction executed on average. In contrast, the number of instructions executed per clock cycle of the machine ($instrs/clock$) is largely a function of the microarchitecture of the machine. This component factor of overall machine performance is naturally termed *instructions per clock* (IPC) and is largely determined by the microarchitecture of the machine. Different microarchitectures can be devised that allow for one or more instructions to execute simultaneously within the hardware of the machine. Generally, simpler microarchitectures allow for fewer instructions to execute simultaneously and more aggressive designs can accommodate larger numbers of instructions to execute simultaneously. Finally, the number of clocks executed per unit time ($clocks/time$) is largely determined by the technology of the integrated circuit (IC) design process. This includes issues such as choice of semiconductors (for example: silicon, or gallium arsenide, et cetera), choice of transistor types (bipolar, junction field effect, metal oxide semiconductor field effect, et cetera), feature lengths, and doping types and precision. Improvements in IC process technologies has allowed for quicker (shorter) gate delays and correspondingly higher clock rates. These three component factors of machine performance that we have introduced above (given in Equation 1.1) and their primary associations with aspects of a machine design are summarized as follows:

factor	machine aspect
$\frac{work}{instr}$	architecture
$\frac{instrs}{clock}$	microarchitecture
$\frac{clocks}{time}$	technology

Our present work is not concerned with either the design of computer architectures (including instruction set architectures) or with integrated circuit technologies. Rather, our focus is on exploring new microarchitectures for improving execution performance. It is therefore the second component factor term of Equation 1.1 that is our primary concern. It is this factor that we try to increase in our present work. The first term of Equation 1.1 is completely out of our control as we are not attempting to design a new instruction set architecture. Likewise, the third term of Equation 1.1 is significantly out of our control as we are not concerned with the specific technology used to implement any microarchitectures that we might consider. But it is second term of Equation 1.1 ($instrs/clock$ or IPC) that is largely addressed by our microarchitectural contribution. We do note in passing that different microarchitectures can take different approaches to how much logic is involved in each clock period of the machine. One microarchitecture can feature a design decision to have a large amount of serially dependent logic be evaluated in each clock period, while

another microarchitecture can decide to limit the amount of serially dependent logic for each clock period. Although this is an important trade-off for computer designers, our present work is of a more introductory level and therefore specific decisions about how much serially dependent logic should be implemented for each clock period is somewhat beyond the scope of our present work.

We now will examine methods and machine design approaches of increasing IPC as a means to increase overall machine performance.

1.2 Instruction level parallelism

One common approach for increasing the execution performance of a program is to have the machine execute several instructions simultaneously. This method is termed *instruction level parallelism* (ILP) since the computing abstraction level that is being exploited through parallelism is at that of the program instructions themselves. This idea also conveys the granularity of the parallelism being exploited. A courser grained level of parallelism could be to independently execute whole sections of a program on separate processors. A finer grained level of parallelism might be to execute component parts of a single instruction (given a sufficiently complicated instruction) simultaneously in hardware. The number of instructions that can be executed per single clock on a given computing machine (the IPC of the machine) is largely determined by two major factors. The first is a property of the compiled binary program itself and is limited by how many independent instructions are readily available to be executed in parallel. The second major factor is due to the microarchitecture of the machine and how aggressive it is about finding multiple suitable instructions within the executing program and executing them in parallel.

In our present work, we take as a given the compiled binary programs that are presented to be executed on our machine. The task of the microarchitecture (and our work) is to find suitable instructions within the instruction stream of the executing program that can be executed in parallel. The process within the microarchitecture of finding and executing instructions from the binary program instruction stream in the order and simultaneous number of its own choosing is termed *dynamic instruction scheduling*. Although both the programmer (to some extent) and the compiler (through its scheduling of instructions) can very substantially affect the number of instructions that can easily or readily be executed in parallel by modest hardware, these methods are beyond the scope of our work. Rather we concern ourselves with the task of the machine in finding the suitably available instructions in the given binary program that might be exploited for performance gains through simultaneous execution.

We take a moment to note here that there are processor architectures [119] (like the iA-64 family of machine architectures [54, 76]) that feature an ISA that supposedly alleviates (being one of their primary design intents) the processor microarchitecture from having to dynamically

find instructions to execute in parallel. However, for our present purposes, these architectures just represent one choice for an ISA along a continuum of such choices and in this sense are just another general (not special) ISA that we assume is a given for which we need to implement through a microarchitecture. These sorts of ISAs represent an attempt to have the compiler (or programmer) statically determine the order and simultaneous number of instructions that can or should be executed by the microarchitecture. As a consequence, these ISAs can be characterized as allowing or facilitating *static instruction scheduling*. It should be further noted that although an ISA may have been designed to facilitate static instruction scheduling (usually by the compiler), it can still be dynamically scheduled with a suitable microarchitecture. Again, this shows that for our purposes, the exact nature of the ISA can be considered an orthogonal design parameter to the microarchitecture of the same machine.

Another more recent advancement with processor development has been the introduction of multiple processors on the same silicon integrated circuit chip. An example of this approach is that of the Power-5 processor from IBM. [65] The Power-5 is also interesting because it also embodies an example of multithreading. [144] The DEC/Compaq Alpha EV8 processor [57, 120, 102] was an aborted attempt at multithreading also. However, from a microarchitectural point of view the introduction of either multiple processor cores on the same IC chip or the introduction of multithreading within a single processor core does nothing to address the goal of extracting ILP from a single sequential program. Rather, both of these advancements allow for the simultaneous execution of multiple independent program streams. In the case of the duel core processors, an independent program stream is executed on each of the cores. In the case of multithreading, an independent program stream is executed within each thread, where each thread shares the common hardware resources of a single processor hardware implementation. The fact that the same program might execute in parallel in each case does not address the program of extracting ILP from a single program thread. Instead, each core or thread is just executing a separate instance of the program as might be done in prior convention multiprocessor computers. So although these approaches increase the execution utility of silicon space, they do not represent advancements in speeding up single threaded program execution, which is the goal of this present work.

Finding instructions within the executing binary program that can be executed in parallel is not a straightforward task. Both the order of the presented instructions as well as their data and control dependencies among them constitutes a major factor in the amount of ILP that we might expect to extract with a given microarchitecture. [59] Obviously the numbers of instruction management and execution component resources within any given microarchitecture also is a major factor in the amount of ILP that might be extracted. The hardware resources of any given microarchitecture constitute the structural hazards that serve to limit instruction execution parallelism.

As alluded to already, one of the greatest constraints within programs that limits the number of

instructions that can be easily executed in parallel are the dependencies between the instructions in the given binary program. This matter is briefly introduced in the next section.

1.3 Program dependencies

For the programs that we want to consider (single-threaded integer sequential programs), they generally contain a large number of inter-instruction dependencies that serve to limit the amount of IPC parallelism that can be achieved with any given machine microarchitecture. These dependencies are usually classified into two main categories: control dependencies, and data (or value) dependencies. Although both control and data dependencies serve to limit how many instructions can be easily executed simultaneously, they do so in very different ways.

Control dependencies are caused by the presence of control-flow-change instructions within the binary program. Instructions of this type are generally: jump, call, and conditional branch. In general (depending on the ISA of the given machine) these instructions can change the flow of program execution to either a single other location within the program or to two or more other locations within the program. Further, the locations where program flow is directed might be fixed or static (determined at compile time) or can be computed at execution time (or run time). For those cases where the control flow can be redirected to either a variable location or to two or more locations, they present an especially difficult situation for the underlying microarchitecture. The problem is that the microarchitecture either needs to wait until the new program location is resolved or needs to decide to speculatively execute down one or more speculative paths. One alternative rather than waiting for the resolution of the control-flow-change instruction is to try to predict the target execution location of the instruction and to speculatively execute starting at that target location. Many microarchitectures already employ speculative execution of predicted program paths but generally only one speculative path is pursued at a time. The reason is due to the possible exponential growth of speculative paths as additional control-flow-changes are encountered in the instruction stream. For example, when multiple speculative conditional branch instructions are encountered, execution is often limited to following just one of the two output paths at each conditional branch encountered. This is done because the hardware complexity (management, tracking, and storage of state) of handling more than one output path at each subsequent conditional branch increases exponentially (base two) with the number of speculative conditional branches. Intermediate strategies between just handling one speculative execution path and all of them are also possible. But the handling of more than a single speculative execution path is greatly complicated by the fact that a mere first-in-first-out (FIFO) reorder is not longer adequate for managing the outcome possibilities. In short, the problems associated with conditional branches whether through poor output prediction or with the hardware complexity of managing and storing speculative results,

makes them a difficult programming construct to handle within the microarchitecture. Therefore the hardware handling of all conditional branches tends to be a performance limiter.

One way to mitigate the hardware difficulties associated with conditional branches is to try to limit their occurrence in the original program itself. However this is a programmer or a compiler issue only and beyond the scope of our work. Another means to mitigate the effects of multiple successive conditional branches is to use an ISA that employs an idea called architectural predication to convert code that would need conditional branches into code that doesn't. But this is an architectural solution that is not available for existing ISAs and therefore not available for legacy ISAs (which will be around for a long time to come). A final alternative is to mitigate the effects of conditional branches through predication within the microarchitecture itself. In our present work, we introduce a new way to perform generalized microarchitectural predication. This scheme effectively converts control-flow dependencies into something similar to data flow variable dependencies. Our microarchitectural predicate variables also function in a way very similar to architectural data variables, along with the attendant dependencies that are amongst them. Our microarchitectural predication scheme also allows for a more scalable and distributed microarchitecture implementation than with previous approaches. We also introduce a way to handle our new control-flow predicate variable dependencies in a generalized and efficient way, within the microarchitecture, similar to how architectural data dependencies are handled. Indeed the present work handles both of these types of dependencies (control-flow and data) in a similar way.

Most conventional microarchitectures are also limited in that they can only speculatively execute until they encounter an instruction with a true data dependency from a previous instruction. Although it is possible to predict data values (in a somewhat similar way in which control flow predictions are made) the burden to the underlying machine hardware needed to track multiple instructions that have had their data value input dependencies predicted becomes elaborate and resource intensive. We also introduce a new approach to manage the tracking of speculatively predicted input data dependencies for instructions. Our approach even allows all instructions in-flight to manage speculative input data dependencies in a way that both maximizes flexibility about which instructions can be executed speculatively and also in what sequence they can be speculatively executed.

Although the major part of this work involves itself with methods to organize and manage hardware microarchitectural resources for program ILP extraction, we first want to consider the potential ILP of a program with idealistically relaxed machine constraints. This will provide us with some expectation for what might be possible with future microarchitectures; ours being among those future microarchitectures! This is briefly addressed in the next section.

1.4 Potential Instruction Level Parallelism in programs

An important question to consider before undertaking an extensive exploration into ILP extraction methods within the microarchitecture of a processor is to consider how much ILP might be present in the workloads that we want to address. Several studies into the limits of instruction level parallelism have shown that there is a significant amount of parallelism within typical sequentially oriented single-threaded programs (for example the SpecInt-2000 benchmark suite). The work of researchers including Lam and Wilson [78], Uht and Sindagi [148], and Gonzalez and Gonzalez [39, 42] have shown that there exists a great amount of instruction level parallelism that is not being exploited by any existing computer designs.

The work by Lam and Wilson is especially comprehensive and interesting in that several aggressive approaches to parallel execution were explored. Although they did not address execution approaches where general data dependencies are broken (data dependent instructions being speculatively executed) they did explore very aggressive speculative parallel execution through the breaking of control dependencies. For conditional branch instructions, its associated data dependency (what the branch instruction is making its outcome decision on) was effectively broken by the speculative execution of subsequent instructions through branch prediction. Some of the more interesting results of their study are shown in Table 1.1:

Table 1.1: *Some results of ILP extraction from the Lam and Wilson study.* The amount of ILP extracted from integer serial benchmarks is shown for three types of various control speculation and execution techniques. A harmonic mean across all benchmarks for each type of machine execution type is also given.

benchmark	BASE	SP-CD-MF	ORACLE
awk	2.85	41.88	242.77
ccom	2.13	18.05	46.80
eqntott	1.98	225.90	3282.91
espresso	1.51	402.85	742.30
gcc(cc1)	2.10	66.29	174.50
irsim	2.31	45.86	262.42
latex	2.71	18.65	131.69
H-MEAN	2.14	39.62	158.26

One has to see the results to really believe them, at least in this case ! Table 1.1 shows the amount of ILP extracted from several integer serial sequential programs (taken from the SpecInt suite). Three different machine execution models are presented that were explored by Lam and Wilson. These machine models are labeled: BASE, SP-CD-MF, and ORACLE. All instructions were idealized to execute in a single clock and so the ILP essentially represents the resulting

maximum IPC possible from the given machine. The BASE machine represents an execution model where no instructions beyond conditional branches are speculatively executed, but instructions can otherwise executed entirely in parallel once observing the necessary data dependencies. Instructions may still execute in parallel across conditional branches but only if the conditional branch is resolved (but not all prior instructions were). This machine is actually more conservative than many current machines in that no speculative execution beyond branches is done, but it features unrestrained machine resources otherwise. The SP-CD-MF execution model represents a machine that allows for any type of control-speculative execution (all data dependencies are observed) except that multiple output paths of conditional branches are not executed in parallel. In effect, an instruction cannot execute until all of its mispredicted control dependence branches are resolved. Finally, the ORACLE machine represents entirely unrestrained control-flow speculative parallel execution but with also observing instruction data dependencies.

Although the results vary substantially from program to program, they are still enormously encouraging and instructive. The BASE machine is interesting in that it shows that without conditional branch prediction, ILP extraction expectations remain relatively low. Note, however, that present machines that do employ conditional branch prediction still do not generally achieve the performance of the BASE machine (an harmonic mean ILP of 2.1 for BASE while current machines often perform at less than 1.0). The SP-CD-MF machine represents something more along the lines that we are pursuing. The harmonic mean extracted ILP (39.6) is way beyond anything achieved with existing machines. Although this is an upper bound for its very relaxed execution restrains, it shows what might be possible through very aggressive machine execution techniques. The ORACLE machine (harmonic mean ILP of 158.3) shows what might be possible with a machine that speculatively executes down both outcomes of conditional branches simultaneously. Our own presented work includes the introduction and exploration of two aggressive microarchitectures that each attempt to allow for the relaxed restraints of the SP-CD-MF and ORACLE machines. However, unlike the idealized machines of this study, our machines have limited machine resources thusly presenting structural hazards to execution.

In the next section we outline how we handle speculative execution in a new and general way. Our new approach maximizes opportunities for speculation while also allowing for a scalable and distributed approach to the management of intermediate speculative operands and other state.

1.5 Speculative execution and resource flow computing

Speculative execution has proven to be enormously valuable for increasing execution-time performance in recent and current processors. The use of speculative execution provides a powerful

latency-hiding mechanism for those microarchitectural operations that would otherwise cause unacceptable stalls within the processor, such as waiting for conditional branches to resolve or for memory reads to be fulfilled from the memory hierarchy. Further, in order to extract ever larger amounts of instruction level parallelism from existing programs (generally quite sequential in nature) over many basic blocks, much more speculative execution is usually required. However, the complexity of implementing speculative execution is substantial and has been a limiting factor in its evolution to more aggressive forms beyond control-flow speculation.

Most existing implementations of speculative execution focus on conditional branch prediction and the subsequent speculative execution of the instructions following those branches. Generally, only one path following a branch is followed although multiple successive branches can be predicted. Speculative instruction results are stored in microarchitectural structures that hold those results as being tentative until they can be determined to constitute the committed state of the program being executed. If a predicted branch is determined (resolved) to have been mis-predicted, any speculatively executed instructions need to be squashed. This generally entails the abandonment of any speculatively generated results as well as the purging of all currently executing speculative instructions from the machine. The management and sequencing of existing speculative execution is already moderately complicated. This complexity has limited or discouraged the use of more advanced speculation techniques such as value prediction, multipath execution, and the retention of speculative instructions that may still be correct after a misprediction. Further, as clock speeds get higher and pipeline depths get larger the performance penalty of squashing speculative instructions and any associated results gets undesirably larger also.

We present a microarchitectural approach towards handling speculative execution in a more general and uniform way. Our approach attempts to generalize the management and operational issues associated with control-flow prediction, value prediction, and the possible re-execution of those instructions that have already been fetched and dispatched. Moreover, in order to accommodate the much larger number of speculatively executed instructions needed in order to extract more instruction level parallelism from the program, a strategy for scaling a microarchitecture in terms of its component resources needs to also be formulated. Our microarchitectural approach also lends itself towards resource scalability through manageable spatial distribution of machine components. This last goal is likewise realized through the rather general and uniform way in which instructions and operands are handled.

Our microarchitectural approach is termed *Resource Flow Computing* and centralizes around the idea that speculative execution is not constrained by either the control flow graph or the data flow graph of the program, but rather by the available resources within a representative microarchitecture. In other words, only structural hazards within the microarchitecture serve to limit the amount of execution (either speculative or not). Further, we elevate the importance of

instruction operands (whether they be control or data) to almost the level of an instruction itself. Operands are enhanced with additional state that allows for a more unform management of their flow through the machine. This philosophy of execution allows for a large number of simultaneous instruction executions and re-executions as can be sustained on the available machine resources since executions are allowed to proceed with any available feasible source operands whether predicted or not. The idea is to first speculatively execute any pending instruction whenever any suitable machine resource is available and then to perform successive re-executions as needed as control and data dependency relationships are determined dynamically during execution.

The present work presents the basic concepts employed in the implementation of a Resource Flow microarchitecture. We first introduce resource flow computing apart from a complete microarchitecture to orient the reader to the basic ideas and most basic components. The most central and basic component of resource flow computing is an enhanced and modified version of a reservation station. Through this component, and the interactions among several of these components, a preliminary introduction to resource flow computing is presented. We then introduce the first of two microarchitectures employing the basic resource-flow execution model. The first microarchitecture is most similar to conventional super-scalar microarchitectures but embodies the increased instruction execution speculation flexibility allowed for through the resource flow execution model. Our first microarchitecture presented is termed *OpTiFlow*. Its name is an attempt at abbreviating the idea of *Operand Time Flow*, an idea which forms a central part of the Resource Flow execution model as well as microarchitectures built around that execution model. Our second example microarchitecture also employs the ideas of resource flow execution but also introduces the idea of hardware scalability and distributed hardware circuit layout. The introduction of distributed and scalable numbers of components allows for a microarchitecture that can employ many more basic computing elements (resources) than most conventional microarchitectures. This increased size in the numbers of computing resources may serve as the basis for very large execution performance increases in the future as the ideas of resource flow computing are tuned and refined in subsequent future microarchitectures. This second microarchitecture that we introduce is termed *Levo*, and it is so named in tribute to its principal proponent. We do not confer any additional associate with the name in this present work.

1.6 Physical hardware scalability and increased hardware resources

One of the difficulties in achieving increased execution performance is the physical realization of increased numbers of processor components that can be allocated to the task of more aggressively executing large numbers of instructions in parallel from the target program. Conventional super-scalar machines currently employ centralized hardware components that can serve as constraining

factors for increased execution performance. The use of centralized components can often limit their size (in terms of numbers of elements within them), and this in turn can limit performance due to the inability to achieve a larger amount of parallel execution through the efficient use of the increased numbers of elements within the components. Some of the conventional processor structures that are centralized include things like the architected register file, the instruction window or a register update unit, and the reorder buffer. The problem with simply scaling up the number of slots or elements within these components, in order to facilitate a much larger amount of microarchitectural execution parallelism, is that the size and numbers of buses (or ports) that need to interact with these various components also increases. In fact, the number of bus interfaces would need to increase to a degree where they are simply physically unrealistic to implement. At present, more than approximately eight simultaneous ports to these various structures is not realistic. Further, even if a design was constructed where there was a greater number of interface ports above something approximating eight, the layout and interconnection complexity alone would now tend to be the overwhelming limiting factor, leading to increased propagation delays and also a large amount of silicon space devoted to nothing but interconnection buses and switching logic. Further, often a large sized centralized component can represent a critical timing path in the design due to the signal routing difficulties associated with accessing it. [98] Therefore in order to substantially increase processor hardware resources beyond current amounts, some new strategy for interconnecting the basic elements to carry out speculative instruction execution needs to be introduced.

This work introduces just such an approach towards the reorganization of the basic required execution resources so as to facilitate a distributed layout approach to the components thus allowing for a larger number of components to work together on a larger number of target program instructions in parallel. The distributed management of speculative execution results and how these are managed to converge properly to construct the committed program execution state (how the program state is supposed to evolve in the abstract architectural perspective) constitutes the most important contribution of this work. Some of the microarchitectural framework for realizing a distributed hardware microarchitecture is already embodied in the hardware components that we present that make up the resource flow execution model (an execution model that is only structurally constrained). This will be evident when we show how distributed scalability is achieved while essentially retaining the existing components that facilitate resource flow execution. However we also introduce in this work two additional key microarchitectural ideas that facilitate the goal of realizing distributed hardware scalability.

The first of these new ideas is a way to perform control-flow dependency analysis and management in a distributed way. Our idea (presented in Chapter 7) is somewhat similar to architectural (or ISA) instruction predication but instead is entirely implemented microarchitecturally. That is,

it is not visible at the architectural level of abstraction. In other words, it is a feature of the machine and not of the architecture or ISA. However, it does not preclude the implementation by the hardware machine of instruction predication within the ISA either. This allows for the emulation by the hardware of existing ISAs that may either employ or not employ architectural instruction predication. Prior microarchitectural predication (of which our present work is substantially based upon) required centralized hardware resources. This limited its physical access by the rest of the machine and its hardware size scalability. Our new approach is implemented within the microarchitecture in a distributed way so as to facilitate hardware component scalability rather than hinder it. Like the previous generalized microarchitectural predication scheme that we have enhanced, our new scheme effectively converts control-flow program dependencies into something very similar to data dependencies. This conversion of control-flow dependencies actually facilitates (as we will show in a subsequent chapter) the management of control-flow dependencies that has not been possible with existing microarchitectures not using some form of generalized microarchitectural predication (of which there presently are none).

The second of these new ideas is that of an operand forwarding repeater unit. This is a new physical hardware component that serves the role of repeating the propagation of intermediate result operands (either of committed architectural significance or of speculative significance) from one element of instruction execution to subsequent ones. This type of component allows for interconnecting operand forwarding buses to be electrically isolated so as to limit their length and to minimize associated propagation delays so as to fit desired design clock periods. By electrically repeating interconnection buses, we also facilitate the physical scalability of the entire machine, allowing for increased numbers of components that would not otherwise be possible with unrepeated electrical buses. However, beyond just providing electrical repetition of buses, our scheme provides for the local or nearby storage of many possibly required operands for the physical structures that are executing local instructions. This is entirely similar to the idea of memory caches, but which are instead applied within the execution window of the microarchitecture itself. This is consistent with the idea that if the execution window itself eventually becomes a sort of distributed array of execution areas, clusters, or stations (the idea of which is part of what we introduce), then local operand caches are a logical solution to the problem of repeatedly acquiring operands from distance physical locations within the whole of the processor. We further discuss how these operand caches (physically a part of the repeater units) handle operands of register and predicate type (a type introduced through generalized microarchitectural predication), as well as the more convention memory operands.

1.7 Simulator development

Since our present work involves exploration into microarchitectures that are very different than any existing microarchitectures, we also create and introduce our own new simulator suitable for the task. Since all existing superscalar microarchitectures are relatively simple as compared to what we introduce, a new and more expansive microarchitectural simulator is needed. The majority of processor research efforts (over fifty percent) has largely relied on a fairly simply simulator named SimpleScalar [6]. This simulator models, in a basic way, a rather conventional and simple four stage pipelined superscalar machine. Although different levels of machine model realism is available with the suite, the only two machine models provided in the suite that somewhat resemble actual machines use either a register update unit or a reorder buffer for holding speculative results. Although this simulator (and others like it) has been very useful for a large variety of microarchitectural research, it is entirely inadequate for the substantially increased dynamics of either the resource flow execution model idea itself or either of the two representative microarchitectures that we introduce later in the present work. For this reason, we have created a new machine simulator that allows for the modeling of a rather more arbitrary collection of sequentially clocked hardware components that is not possible with most of the current and available machine simulators. Our new simulator also more closely models the sequential state dynamics of actual state machines used within our machine components.

Our new simulator forms a basic platform for the evaluation of our new microarchitectures. More specifically, it allows for the testing of the new hardware components that we introduce in the present work. More details on the nature of this simulator is presented in a subsequent chapter.

1.8 Scope and contribution of the work

The present work does not address every conceivable way in which computer performance can be improved. Two main areas of potential execution performance are not the scope nor focus of this work. The first of these is that we assume an instruction set architecture as an input to our exploration process. This means that the techniques that we explore should be able to be applied to any existing ISA. This is far from an unrealistic expectation for much new computer performance enhancement research. Rather, the commercial need to maintain binary compatibility with legacy ISAs is an extremely important goal of most research in computer performance improvement techniques. Secondly, the present work is not concerned with the silicon processing technologies of improving gate propagation delay times and the problems associated with minimizing signal wire propagation times. These are integrated circuit design issues that are outside of our scope of investigation. Rather, our work is concerned with new ways to organize the microarchitecture of a machine

in order to better manage more speculative execution while minimizing both hardware resources and also allowing for increased physical scalability of the numbers of hardware components.

Another area not specifically addressed with our present work is that of power consumption. Although obtaining good (low) ratios of power consumed per unit computational work accomplished is an important market design point, that is not our primary focus. Rather, our goal is to explore ways to maximize performance through new organizations of the hardware microarchitectural components of the machine. This goal applies both for a fixed amount of hardware resources as well as increased numbers of hardware resources (also facilitated in part by our present work). We don't hesitate to note, however, that if a machine of some fixed hardware resources (measured in a basic way through something like for example the total number of transistors in the design) can have its execution performance improved through a novel microarchitectural organization, then a smaller sized machine might be constructed that achieves the same execution performance as the baseline machine but with possibly reduced power consumption.

The main contributions of the present work are summarized as:

- the development and explanation of the resource flow execution model
- the development and description of the primary hardware component used in all of the resource flow microarchitectures that we introduce
- the development of a new microarchitectural simulator that accommodates the intricate dynamics of our microarchitectures that we introduce and evaluated
- the development and description of the first of two microarchitectures using the resource flow execution model; this microarchitecture most closely resembles a more conventional superscalar machine in that it is not entirely suitable for physical scalability
- an introductory performance evaluation and characterization of the first (simpler) of our two proposed microarchitectures using the simulator that we have developed
- the detailed development and description of the operand repeater unit that we have developed
- the development and description of a second microarchitecture employing the resource flow execution model that is physically scalable due to the addition of operand repeater unit components and the additional of generalized microarchitectural instruction predication
- a preliminary performance evaluation of the second of our proposed microarchitectures embodying the resource flow execution model

1.9 Organization of the present work

Chapter 2 provides a background on the basic microarchitectures used for existing dynamically scheduled superscalar machines. With this background of how existing machines operate, we are sufficiently oriented to apply the basic ideas of these machines in a new way in order to achieve the design goals that we want. We also present a brief summary of some previous work by others in the area of instruction level parallelism through the use of microarchitecture innovation. Chapter 3 introduces the resource flow execution model along with an introduction of a primary machine component to facilitate that execution model. Chapter 4 introduces the first of our two proposed microarchitectures that embodies the resource flow execution model. Chapter 5 provides some information on the simulator that we have developed in order to evaluate and characterize the first of our two new microarchitectures. Chapter 6 presents some preliminary performance results from the first of our proposed resource flow microarchitectures. Chapter 7 provides an introduction to a new generalized microarchitectural instruction predication scheme that facilitates the physical scalability of resource flow microarchitectures. Chapter 8 presents some characterization of temporal operands, which in turn provides the clues used to develop our operand repeater unit ideas. We also present a hardware introduction to our operand repeater units for the repetition of register, predicate, and memory operands. Chapter 9 evaluates the suitability of our operand report unit scheme for achieving physical scalability. More specifically, this chapter presents register and memory access interval characterization data that confirms the feasibility of our approach to physical scalability. Chapter 10 introduces the second of our proposed resource flow microarchitectures and highlights how it is physical scalable to accommodate larger numbers of hardware component resources. Finally Chapter 11 summarizes and concludes the present work.

Chapter 2

Background

In this chapter we first present the basic operation of some prior (and existing) microarchitectures that exploit instruction level parallelism (ILP) from the executing target program and which feature dynamic instruction execution scheduling. This is appropriate since our goal is to advance the state of the art with respect to this type of parallelism. Many of the examples are quite historic and essentially span most of the period of development for those ILP-extracting microarchitectures that also feature dynamic instruction execution scheduling (as opposed to strictly static instruction execution scheduling). With these microarchitecture examples we orient the reader to many of the basic machine operations required for program instruction execution on an ILP oriented machine are performed. Many of the most basic operational elements of all computing machines are similar and these similarities will be highlighted using the examples.

While presenting a number of different microarchitectures, and variations on them, in the following sections we note that any of these microarchitectures can be used to implement the same computer architecture (including its instruction set architecture as well). Of course, for all cases of different microarchitectures, the programmer always expects his program to execute according to the architectural description of the machine provided by the computer designer. This is the essence of the contract between the computer designer and the user programmer (or compiler writer). Although the microarchitecture for any given computer architecture may be different in different machines, there is a guarantee from the computer designer to the user programmer that their binary programs will still run on any of the possible machines implementing the same architecture.

For completeness, one caveat by the computer designer that technically violates the contract with the programmer (and which is not uncommonly exercised) is that generally only the user mode architecture is strictly compatible between different implementations of a computer architecture. The user mode architecture is that part of the whole architecture that is visible to user mode programs running on the computer. Examples of programs or program code that would not strictly

be user mode code would be code entities like devices drivers and the operating system itself. Often, these codes are executed in a machine context usually referred to as kernel mode, supervisor mode, or master mode (different vendors use different names). These latter examples of program codes might need to be modified between different machine microarchitectures in order to accommodate incompatible kernel-mode architectural changes from one microarchitectural implementation to another. Some examples of architectural changes that only affect kernel mode execution might be the way in which a certain machine function like a programmable timer is programmed. Another example kernel mode difference between different machine implementations could be the number of translation look-aside buffers present in the memory management unit. Architectural differences like these generally require changes in one or more components of the kernel mode program codes, but do not require changes in user mode program codes. Generally when architectural compatibility is a design requirement, it is the compatibility of the user mode architecture that is maintained across diverse microarchitectures.

We present a number of microarchitecture machine organizations in roughly the order in which they were developed. Fortunately, when they are presented in this order there are relatively few changes from one microarchitecture to the next. This presentation approach will help to point out the similarities and evolution of these machines, and hopefully will be useful for understanding our own microarchitectural introductions presented in subsequent chapters. For brevity, we do not present the simple microarchitecture that employs strict sequential execution of successive instructions, but neither do we disparage that simple microarchitecture. We assume that the reader is already familiar with that model as it is generally what is presented to the user programmer to explain the precise architectural behavior of a given machine. Many simple processors are still purposefully designed to execute the architectural instructions of the processor in a strictly sequential way. The associated microarchitecture is generally very simple and offers niche advantages such as small silicon space requirements and often a low power per unit work ratio.

The remainder of this chapter is organized as follows. We first present the microarchitecture that initially introduced parallel execution of instructions through the use of separate execution units, each of which only executed a particular class of instructions (a subset of all instructions). We then present the first microarchitecture to allow for the renaming of registers thus avoiding false register dependencies in the program code given us to execute. Next we introduce a microarchitecture that allows for speculative execution through the prediction [13, 170, 171] of control-flow instructions. Speculative execution is facilitated through the addition of an instruction reordering component. Following this we introduce another incremental variation that allows for reducing the logic complexity of the reordering component previously introduced. This is followed with an incremental improvement that increases the number of instructions in-flight within the machine beyond the number of instructions that can be executing simultaneously. This improvement also increases

the utilization of machine execution resources that may have been idle previously. We are now ready to show microarchitectures that very closely resemble many of those in current use today. These latter microarchitectures feature a consolidation of similar distributed machine structures into a single centralized component. This change allows for better utilization of resources making up that component. At this point we briefly discuss the idea of injecting more than one instruction into the execution core of the processor in each clock period. We also briefly discuss the technique of pipelining component parts of the machine. next, we show an existing microarchitecture that features an organization (introducing the Register Update Unit) that somewhat foreshadows the primary microarchitectural component that is featured in our own microarchitectural organizations presented in subsequent chapters. We then briefly discuss the idea of only using unified execution units rather than instruction-classed execution units. This last idea is widely used in current processors and is also featured in one of our newly introduced microarchitectures. We will also briefly introduce the idea of Disjoint Eager Execution (DEE). This is a form of multipath execution but represents a different way to allocate machine computing resources towards speculatively executing instructions than what is presently done in existing microarchitectures. This idea (DEE) is used in the second of two microarchitectures that we introduce later.

Hopefully, seeing the evolution and organizational variations in the microarchitectures presented provides an overall orientation for our extension of these existing machine organizational ideas to our own. In concluding the chapter we briefly discuss some prior work with some research microarchitectures that have also attempted to address the problem of extracting instruction level parallelism from substantially integer sequential programs. Several of the research ideas used in these research microarchitectures is picked up and used within our own microarchitectural work (discussed in subsequent chapters).

2.1 Parallel execution of instructions

We start by presenting the first microarchitecture to execute instructions in parallel. This microarchitecture was that of the Control Data Corporation (CDC) 6600 first introduced in late 1963 [139, 20]. As an aside, this machine is considered to be the first supercomputer ever built. We only focus on a small part of the operation of this machine. Our goal is to understand the contribution to dynamic instruction scheduling that was introduced with this machine. Therefore we ignore or otherwise simplify a substantial part of the operation of this machine in order to focus on how it managed the execution of its instructions. A very simplified diagram of the central execution part of this microarchitecture is shown in Figure 2.1. This figure only presents the central execution core of the main processor of the CDC 6600. More specifically, for our present purposes, we are not concerned with the way in which memory operations were handled (although this was also an

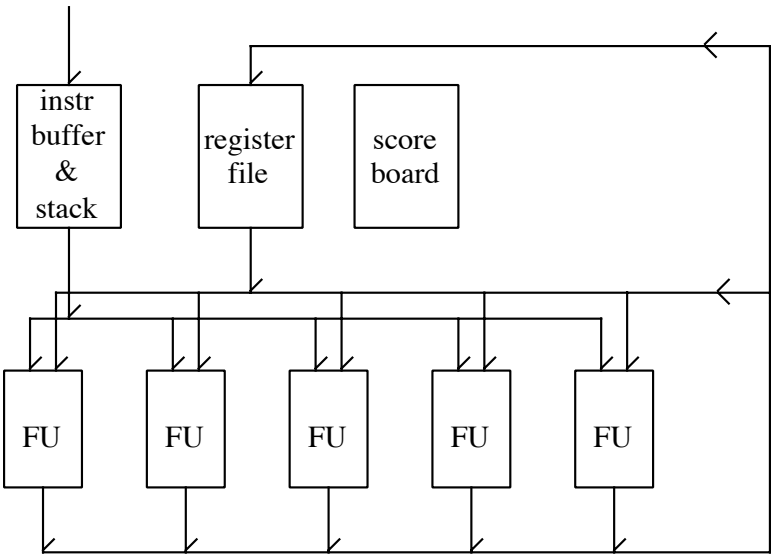

Figure 2.1: *Simplified schematic of the execution core of the CDC 6600 central processor.* Shown is a very simplified schematic of the instruction queue, architected register file, and five of the function units of the central execution part of the central processor of Control Data Corporation (CDC) 6600 computer. This was the first computer to execute instructions in parallel.

interesting feature of this microarchitecture in its own right). Our main interest is how multiple instructions could be executed in parallel while managing the architected registers involved. This machine had both a central processor and ten peripheral processors (themselves implemented in an exceedingly novel and fascinating way). The nature of the peripheral processors are not our concern for our present purposes and is not discussed further here. However the central processor of this machine introduced a number of interesting features that served as the beginning of a large number of dynamically scheduling microarchitectures in the decades that followed. Although Figure 2.1 only represents a brief outline of this microarchitecture, it serves our purposes as a starting point for the more interesting microarchitectures that followed after the tradition of its basic design philosophy.

As can be seen from the figure, this machine featured the introduction of the execution *function unit* (FU). Each function unit served to execute a certain class (a subset) of the instructions implemented in the machine architecture (or more specifically, its instruction set architecture). With the inclusion of several of these units, multiple instructions could execute simultaneously, each instruction being executed in a separate function unit. It should be noted that this arrangement with multiple instructions executing simultaneously is different from that of simply pipelining the execution of instructions. With pipelining, although two or more instructions may be executed

simultaneously (depending on the number of pipeline stages), the simultaneous instruction execution is more likened to overlapping execution since all of the instructions eventually pass through the same hardware pipeline stages. Rather with the adoption of multiple execution function units, instructions of differing instruction classes are assigned to entirely specialized function units, which do not generally share hardware between each other.

We note in passing that two other machines were introduced by the same manufacturer just after the introduction of the CDC 6600. Both of these machines implemented the exact same user-mode architecture (including the central processor instruction set architecture). These other architecturally similar machines were the CDC 6400 and the CDC 6500. These latter two machines did not feature multiple execution function units but instead employed what was termed a *unified* execution unit. A unified execution unit was one that could execute every instruction of the machine's architecture. This is just one of many examples in history where several architecturally similar or identical machines have been introduced commercially while implementing radically different microarchitecture designs. It is interesting to note that according to the designers of these machine, the amount of logic needed to implement all of the function units of the CDC 6600 was less than the amount to implement a combined unified execution unit.

The central processor of the CDC 6600 machine featured an instruction buffer and stack. This component served to buffer fetched instructions before instruction issue and also to store instructions already issued for possible use later. This component is shown in the upper left area of Figure 2.1. This component served as both a staging area for the issuing of instructions for execution as well as something of a specialized cache for control-flow changes. Unlike a conventional cache, the instruction stack acted as a FIFO of previously issued instructions and served to capture program loops if a subsequent control-flow change was made to an instruction that was already in the stack. The need to buffer fetched instructions was due to the fact that the issuing of instructions for execution no longer occurred at a regular easily determined rate. In fact, the entire rate of execution was no longer easily determined due to the fact that various instructions from the program instruction stream could now be executed in parallel to differing degrees depending on the program order of instructions, the data dependencies between them, and the incidence of memory bank conflicts for parallel memory accesses that was possible. Therefore the rate of instruction issue was now no longer easily determined. Note that in this machine, there was no distinction between the idea of an instruction being dispatched and it being issued. Although the nature of what is termed instruction issue in this machine more closely resembles what is termed instruction dispatch in subsequent machines, we continue to use the term *issue* in this discussion since that is what the original designers called that particular operation.

Function units are shown in Figure 2.1 along the bottom of the figure and they are labeled with the abbreviation FU. As shown in the figure, buses allowed for the transfer of instruction operation

codes to pass from the instruction buffer component and each of the function units. Buses were also provided to transfer register operands from the architected register file (shown near the top middle portion of Figure 2.1) and the function units. The register file of this machine only contained architected registers. This means that these registers only contained values that were committed to be a part of the executed program state. Also shown in the figure (near the middle top of the figure) was a component called the *scoreboard*. This component contained both state and logic that managed the issuing of instructions to the function units. this component served as the heart of this machine and was primarily responsible for the very dynamic machine operation and the scheduling of instructions to be executed by the function units. The state contained inside the scoreboard component consisted of the names of the registers needed by instructions already issued to function units (if any), which registers would be needed by the next instruction in order to be issued, and which function unit was generating the next register value for each of the architected registers (if any).

The CDC 6600 introduced a total of ten function units of eight basic function execution classes. These function unit classes along with the number of each are shown in Table 2.1.

Table 2.1: *The function execution unit classes and number each in the CDC 6000 processor.* Presented are the function execution classes and the number of each for the function units that were employed in the microarchitecture of the CDC 6600 processor.

function class	number of units
BRANCH	1
BOOLEAN	1
SHIFT	1
ADD	1
LONG_ADD	1
MULTIPLY	2
DIVIDE	1
INCREMENT	2

Note that although there are only eight different function unit classes, there are a total of ten function units. The units `MULTIPLY` and `INCREMENT` are duplicated with two units of each. For those curious, the `ADD` unit performed floating point additions and subtractions while the `LONG_ADD` only performed integer additions and subtractions. The `BOOLEAN` unit performed all integer logical operations with the exception of bit-wise shifts, which occurred in the `SHIFT` unit. The `INCREMENT` unit is used to perform integer additions and subtractions for memory address computations. This unit is also used to initiate memory load and store operations. For completeness, some branch instructions of the ISA also performed arithmetic operations (comparisons) and would actually

require both the branch unit and a corresponding arithmetic unit before it could be issued for execution. Note that although the machine had ten function units total, only five are shown in Figure 2.1. The FUs in the figure are not differentiated, but that is not important for our discussion.

A very interesting feature of this machine architecture is that memory loads and stores are initiated as side effects of performing certain "increment" instruction operations where the result of the instruction goes to one of a special set of the ISA registers (known as the "A" registers, named A0 through A7). When five of eight special address registers (A1 through A5) of the architecture are the target of an "increment" operation, a memory load is initiated to the corresponding address contained in the targeted A register. When two other registers of this special register set (A6 and A7) are the target of an "increment" operation, a memory store is initiated to the corresponding memory address. In both cases, the memory load or store occurs to the memory locations specified by the final result held in the corresponding address register after the update of its value by the execution of the increment instruction. Also in both cases, the memory operand is loaded to or stored from another register of a special group of ISA registers known as the "X" register set, named X0 through X7. In each case, an operation on an address register A_n results in the corresponding operation on register X_n. Each of the five memory loads and two memory stores can occur in parallel to or from memory. Some observers of this microarchitecture have identified these parallel memory operations as constituting additional function units. However, our presentation of what constitutes a function unit corresponds to what the designers called function units and so for our purposes we do not consider these parallel and independent memory operations to constitute additions units over what we have shown in Table 2.1.

Each function unit can execute one instruction of its class at a time. Each function unit of a given class can execute any of the instructions of that class from the given ISA. In the CDC 6600, the function units were not pipelined in the sense of allowing multiple instructions to flow through them at once. [1] But the instruction pipelining of function units is something that was added in later microarchitectures of this design philosophy. This microarchitecture was only able to issue a single instruction at a time in any given clock. The maximum possible instruction per clock (IPC) rate for both issue and execution was therefore one (an IPC of 1.0). Again, this restriction was relaxed in subsequent microarchitectures that continued in the tradition of this parallel processor design orientation. Later processor microarchitectures that allowed for the issuing of more than a single instruction in each clock period were termed *superscalar* processors. Also, all instructions were issued in-order from the instruction buffer. Instructions were issued sequentially until one or more of the following conditions occurred:

- no suitable function unit was free for the class of the next instruction ready for issue

[1] Interestingly, the designers of the CDC 6600 were the first to comprehensively explore pipelining in the design of computers, and many parts of the CDC 6600 were pipelined for reducing logic propagation delays.

- the register target of the instruction to be issued was the same as that of some already outstanding instruction (a write-after-write hazard)
- a conditional branch instruction was previously issued and had not yet resolved (a control hazard)

The in-order issuing of instructions along with these above conditions is sufficient to guard against violations of any true or false dependencies. Read-after-write (a false dependency) hazards are automatically avoided as a result of in-order instruction issue and the guard for write-after-write hazards. The stall of instruction issuing due to the unavailability of a corresponding suitable function unit represented a structural hazard of the machine. Interestingly, although instruction issue was stalled after the issue of a conditional branch instruction, this allowed for the out-of-order commitment of all other instructions currently being executed. Besides these restrictions on instructions issue, all other combinations of instructions executing in parallel were allowed for. Most instructions of this machine required at least two input operands (at most two register operands and a memory operand). Except for memory store operations, each instruction only generated a single output operand (a register operand). Note that an instruction could be issued to a function unit with less than its required input operands. Required operands that were available at the time of instruction issue came from either the architected register file or the output of some function unit. If an instruction was issued with less than its required input register operands, it would still be issued to a function unit (assuming a suitable one was available) but it would wait for the required operands to become available from whatever function unit was generating it before it wold start executing. Waits for main memory operations to complete (multiple main memory banks could be accessed in parallel) along with the waiting for function units to complete execution of their assigned instructions represented structural hazards of the machine.

Although only a single instruction could be issued to a function unit at a time, no clock delays were required between the completion of a prior instruction and a subsequent one that was waiting for the result of the prior instruction. The ability of a function unit to receive subsequent instructions from the instruction fetch buffer before the execution completion of a current instruction within the function unit was relaxed in following microarchitectures. Of course, duplicated units of the same class allowed for the simultaneous execution of more than one instruction of the same class. Results from function units were transfered on buses (several) to both waiting function units and the architected register file. This bus connectivity is shown at the bottom and right hand side of Figure 2.1. Although the figure only shows a single bus, in actuality, several parallel buses were available for function unit result transfers. Note that a form of register file bypass was possible by the transfer of the result of a function unit directly to the input of another waiting function unit. In this particular machine, an instruction waiting within a function unit could start executing as

soon (in the next clock period) as the register value it was waiting for from another function unit was made available.

It should be noted that, in general, many instructions took more than a single clock period to execute. This meant that a function unit was considered busy for as many clocks as was needed to perform its function. As might be expected, different instructions took varying numbers of clock periods to execute, with simple arithmetic and logical operations being very fast and floating point multiple and divide taking the most of clocks. Although the type and numbers of the function units listed in Table 2.1 may not appear to be an optimal mix from a more modern architectural point of view, it was in fact quite a good mix given the particulars of the machine ISA (a very RISC-like ISA) and the workload expected to be executed on the machine. The expected workload for this machine was almost entirely a floating point scientific one and the mix of function units was designed to expedite instruction parallelism and over all execution performance for such workloads. The design philosophy for this machine was to break the general purpose floating point program being executed into two main and parallel execution tasks. This division of the instructions of the executing program would be performed dynamically through the functional separation of different instruction types to the different instruction-classed function units. One task was to perform the necessary address computations for memory accesses and any control-flow computations and changes. The other main execution task was to perform the core arithmetic operations (generally floating point operations) for the program. It was envisioned that through the design of the ISA and the mix of function units in the microarchitecture that these two program execution tasks could be made to execute nearly in parallel, increasing overall program execution performance. This same basic design philosophy continued in use through most of the subsequent supercomputer designs into the early 1990s.

Note that this machine did not have the ability to either issue instructions after encountering a write-after-write hazard or to speculatively execute instructions. In subsequent sections, we describe machine microarchitectures that relax both of these limitations. Subsequent machine microarchitectures also make a distinction between the idea of an instruction being merely dispatched and that of it being issued to a function unit.

2.2 The reservation station and register renaming

In this section we present a machine microarchitecture that introduced the idea of the generalized *reservation station* (RS). This new structure was an improvement on the microarchitecture of the CDC 6600 (described in the previous section) in that it allowed for the issuing and execution of instructions even after encountering a write-after-write hazard. The architecture that we present here is essentially that of the International Business Machines (IBM) computer named the System

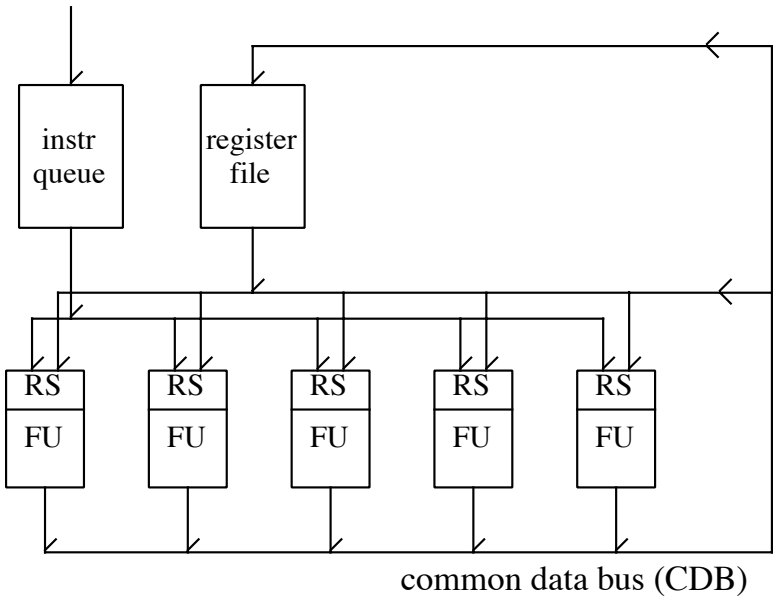

Figure 2.2: *Simplified schematic showing the Tomasulo execution core.* Shown is a very simplified schematic of the instruction queue, architected register file, and five of the reservation stations and function units of the central execution part of a Tomasulo-styled processor.

360 Model 91. [3, 141] A basic outline of the core execution part of the central processor of this machine is shown in Figure 2.2. This machine shared several elements of the CDC 6600. Most notable among these are the existence of several function units. An architected register file was also similar to that of the CDC 6600 in many respects except that it was modified to hold some additional state about where (what function unit, if any) was to create the next value for each architected register. This was similar to some of the state that was stored in the scoreboard of the CDC 6600 except that the goal in this microarchitecture was to allow for the execution of instructions even in the presence of false dependencies (write-after-read and write-after-write).

The primary distinction of this microarchitecture from that of the previous machine discussed is the introduction of the reservation station located at the input side of each of the function units (shown in Figure 2.2 connected to the top of each function unit and labeled with the abbreviation RS). In this microarchitecture, the transfer of a decoded instruction (along with any input operands that might already be available) from the instruction queue (which can buffer up several instructions) to a reservation station is termed *instruction dispatch*. The reservation station serves the function of holding a decoded instruction and of waiting for the proper input operands to arrive for the current instruction from the outputs of other function units that are going to produce those operands.

Instructions are dispatched from the instruction queue to a reservation station without regard for

any register dependencies. Rather, instruction dispatch is only blocked due to the unavailability of a suitable reservation station associated with a function unit. As with the previous example microarchitecture, function units and their associated reservation stations are classed and can only execute those instructions of its own instruction class. Instructions are dispatched in-order and one at a time so the maximum execution rate in instructions per clock is still 1.0 (or an IPC of 1.0). In this microarchitecture, each function unit has associated with it only a single reservation station. However this restriction is relaxed in subsequently developed microarchitectures. We note in passing that with the introduction of the reservation stations, we can double the number of instructions that are in-flight within the machine. As we will see, one instruction can be dispatched to each reservation station while another instruction is being executed in each of the function units. The separation of the reservation station from their associated function units allowed for this increase in the number of instructions in-flight. The advantage of allowing for more instructions to be in-flight (and having been dispatched) is that a greater degree of instruction level parallelism can be exposed in typical program codes. With more instructions being dispatched, the utilization of all function units is more likely increased. Without the addition of the reservation stations, the chances were greater that a function unit had to remain idle due to a structural hazard on instruction class.

Like the previous microarchitecture, instructions can be dispatched to a reservation station either with their proper input register dependencies (in which case they would come from the register file) or without them (in which case they would come from the result of some function unit). For those instructions dispatched to a reservation station without all of their input operands, they wait within the reservation station until their input operands become available. When an instruction is dispatched without one or more of its input operands, special identifying tags (maintained as state within the register file) were dispatched along with the instruction operation code to identify which function units would create the proper values for the appropriate missing operands. Also when an instruction was dispatched, a tag identifying the reservation station where the instruction was dispatch to is updated in the extra state within the register file associated with the name of the output register of the instruction. In this way, subsequently dispatched instructions requiring that same architected register would be dispatched with the proper identifying tag instead for the case when the previous result was not yet available.

Unlike the previous microarchitecture, the logic for determining if an input operand is available is located within each of the reservation stations instead of within the centralized register scoreboard component. Each reservation station contains the state (identifying tags) and logic for determining if its own instruction input operands are available. All output operands from the function units are broadcast to the inputs of all reservation stations and each reservation station snoops for appropriate operand tags to see if the operands it is waiting for became available. Proper input operands are identified through the match of a tag that is assigned to the instruction operand when

it was dispatched to the reservation station. The process of scanning the input operand bus for the proper and necessary input operands is termed *operand snooping*. When an operand match is detected through snooping, the reservation station loads the operand into a register and continues waiting for more either more required input operands or if all input operands have arrived, it transitions into a state where it is ready to have its associated instruction executed. The process of loading a required input operands due to a tag match during snooping is termed an *operand snarf*. When a reservation station acquires all of its necessary input operands and when the associated function unit becomes free (from executing a prior instruction operation), the decoded instruction is transfered to its associated function unit for execution. This transfer is termed *instruction issue*. Once a reservation station is vacated through instruction issue, it is available for receiving another dispatched instruction. Note in Figure 2.2 that function unit output operands can also bypass the architected register file and proceed directly to the inputs of the reservation stations. Note that in this microarchitecture, both instruction issue and instruction execution can occur out-of-order with respect to the proper dynamic program instruction execution order. Results from the function units also update the proper architected register within the register file. The bus that carries output operands from the outputs of the function units was termed the Common Data Bus (CDB) in the IBM 360 Model 91. Subsequent microarchitectures have used a variety of bus arrangements (the simplest of which is a multiplicity of buses) for carrying result operands back to the architected register file and reservation stations.

The primary advantage of the introduction of the reservation station is that now several instructions can be outstanding in the reservation stations where each can be waiting for a different instance of the same architected register. This is facilitated through the assignment of different identifying operand tags for each instruction dispatched to different reservation stations. Through this parallelization of reservation stations and identifying operand tags, a form of register renaming is achieved. Again, this allows for the parallel execution of multiple sequential program instructions even in the situation of there being many false dependencies amongst the instructions being executed. In general, in this microarchitecture, instruction issue must stop after a conditional branch instruction or another control-flow-change instruction with an unknown target address is issued. This is necessary since we do not yet have any microarchitectural capability to track and accommodate speculatively executed instructions. However, this restriction is relaxed in subsequent microarchitectures.

2.3 Speculative instruction execution

In this section we build on the previous microarchitecture by relaxing the restriction of stopping instruction issue after a control-flow-change instruction was issued. This microarchitecture is very

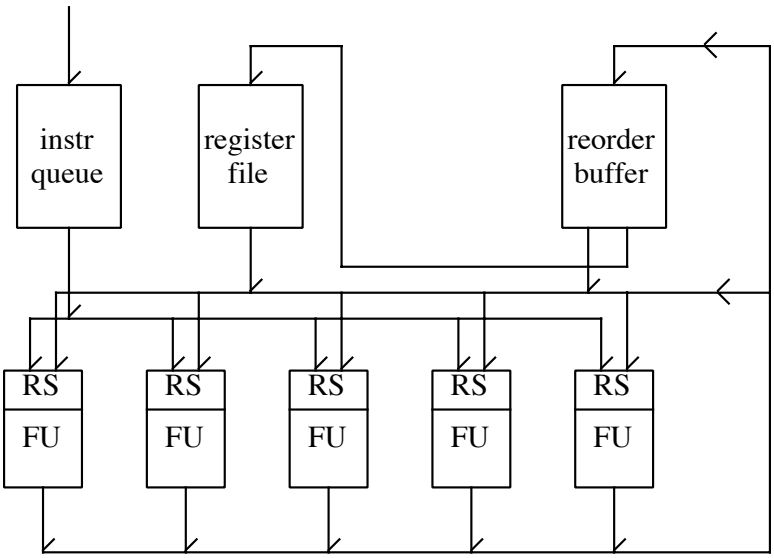

Figure 2.3: *Simplified schematic of a microarchitecture that can perform speculative execution.* Shown is a very simplified schematic of the instruction queue, architected register file, reorder buffer, and five of the reservation stations and function units of a microarchitecture that can perform speculative execution.

similar to the previous one except that a new hardware component is introduced into the register flow path of the machine. This new component is termed a *reorder buffer* (ROB). [128] Output result operands being created by the function units will first pass through this component before being allowed to update the architected register file. This microarchitecture is schematically shown in Figure 2.3. The new reorder buffer component is introduced in the upper right portion of the machine schematic. As can be seen from the figure, all output operands from the function units are first placed into the reorder buffer before passing on (if they ever do) to the architected register file.

The reorder buffer component serves to hold the speculative results of speculatively executed instructions. It is organized as a first-in-first-out (FIFO) structure where each entry contains four primary fields:

- a bit indicating whether the entry is in use (reserved) or not
- the architected name of the speculative register that it will hold
- a bit indicating whether the speculative register value is present or not
- and the speculative value of that register (if available)

Instructions are dispatched from the instruction queue in this microarchitecture similarly as they were in the last one but with a small modification. In this microarchitecture, decoded instructions

are ready to be dispatched when both a suitable reservation station (of the appropriate instruction class) is available, and when an empty in the reorder buffer is available. When both of these conditions occur, a search for the input operands to the instruction is first made in the reorder buffer and then failing that, a search is made in the architected register file. The search of the reorder buffer for possible future operands is made from the last entry placed into it (remember it is FIFO organized) and proceeding towards earlier placed entries. Only if there are no matching entries for a required input register operand is a search made in the architected register file. If an input register operand is available for one or more of the required input operands for the instruction (either a committed value from the architected register file, or a speculative value from the reorder buffer), decoded instruction is dispatched with the corresponding value. However, for those input register operands that are not yet available, a tag identifying which function unit will produce the operand result is dispatched along with the decoded instruction. In addition to having either the actual input operands or identifying FU tags available at the time of instruction dispatch, an additional tag identifying the allocated reorder buffer entry is dispatched along with the decoded instruction. The reorder buffer entry will hold the output register operand value of the decoded instruction after it is executed. Since the reorder buffer is FIFO managed, all newly allocated entries for newly dispatched instruction are logically at the end of the FIFO. Although the reorder buffer is used as a FIFO, it is generally just another random access oriented register array and pointers are maintained to indicate the head and tail of the logical FIFO.

With the introduction of the reorder buffer, now instructions can be dispatched following the dispatch of a prior control-flow-change instruction. This was not possible previously before the introduction of the reorder buffer. Essentially the reorder buffer is the place that speculative results are stored so that only the proper program dynamic state is ever committed to the architected register file. In fact, we now are presented with the reason for the name *architected register file* as opposed to simply *register file*. The architected register file only holds architected state (state both meaning and visible to the programmer), whereas other register files of one sort or another (like the reorder buffer) do not hold architected state but rather speculative state.

A question that may arise with the reader concerns from where are instructions dispatched following the dispatch of a control-flow-change instruction if the result of the control-flow-change instruction is not yet known to the machine (after all, it hasn't even executed yet). The answer is that some prediction of the output control flow path of the control-flow-change instruction is made and that instruction flow path is chosen for further instruction dispatch. Attempts at speculatively dispatching, issuing, and executing more than a single output path is possible but for our present purposes will not be elaborated on further here. There are also many ways (hardware components) that can be used for making output path predictions, but they are not important for our present

discussion. Generally predictor components use one or more tables that maintain control-flow-change history of one or more sorts and this history is consulted at instruction fetch time to make a prediction about the outcome of the control-flow-change instruction. Predictor tables are also updated about actual control flow paths on instruction commitment.

The operation of the reservation stations (snooping and snarfing of input operands) and that of the function units is the same as with the previous microarchitecture. However, as output operands are newly created from the outputs of the function units, they are directed to be placed into the proper reorder buffer entry that was previously allocated at the time of the dispatch of the instruction. Like the previous microarchitecture, output operands are also snooped by waiting reservation stations waiting on input operands.

Register values are only transferred from the reorder buffer to the architected register file when the corresponding instructions can be committed. All output operands for those instructions dispatched after a control-flow-change instruction are stored in the reorder buffer until all preceding control-flow-change instructions are resolved and committed themselves. In the event that a control-flow-change instruction turns out to have been predicted incorrectly, all instruction output operands that reside within the reorder buffer following the control-flow-change instruction are flushed. There can be variations on this basic theme also (like trying not to flush all following entries). Otherwise, if a control-flow-change instruction was predicted correctly, then all instructions following the prior control-flow-change instruction can also be committed for those that have completed execution and are not also following a later unresolved control-flow-change instruction.

One element of the operation of this microarchitecture is worthy of some additional analysis. When output operands are created by the outputs of the function units, they can be easily written into the reorder buffer using the tag that was assigned to them when the corresponding instruction was originally dispatched. The tag was basically just an index into the reorder buffer array structure that identified (addressed) the particular entry to be written. This is a relatively easy operation to be performed in the reorder buffer hardware using the familiar binary decoding logic found on familiar random access memory (RAM) addressed array memory components. However, the instruction dispatch unit cannot access the reorder buffer as simply as the output operand update process (just described) can. Rather, at instruction dispatch, the reorder buffer has to be searched for a matching entry based on the architected name field of the entry, starting from the latest logical entries and continuing onto older entries (remember that the reorder buffer is maintained in a FIFO order). This is a fairly expensive operation to perform in hardware. This is even slightly more complex than the logic needed for the parallel address process for a content addressable memory (CAM). In addition to finding just any entry that matches on the architected register name needed, the logically latest such matching entry is needed. Further, since the reorder buffer is maintained as a logical FIFO, the latest such matching entry is subject to the current state of the

FIFO, and more specifically to where the FIFO head and tail pointers are currently pointing. The complexity of the necessary logic for input operand searching of the reorder buffer has prompted some enhancements to how the whole management of speculative values is maintained. One example of how the logic complexity for input operand acquisition is done at instruction dispatch time is addressed in the next section. Implementation complexity issues will also factor into our own new microarchitectures presented in subsequent chapters.

2.4 Logic complexity reduction for speculative operand acquisition

The problem associated with speculative operand management in general, and input operand management at instruction dispatch time in particular can be problematic for hardware designs requiring both high speed of access from the reorder buffer or in those cases where the reorder buffer is very large (many entries). A larger reorder buffer increases the complexity of the operand searching logic. This is due to the presence of prioritization logic used for determining which entry of all matching entries to choose from when searching on an architected name. The prioritization logic is needed due to the fact that more than one reorder buffer entry may contain the same architected name but only the one in the latest logical order is to be returned. As explained in the last section, this logic is further complicated by the fact that the FIFO pointers used to manage the logical order themselves rotate around the reorder array structure. Finally, the fact that all of the content addressing logic as well as the necessary prioritization logic is all concentrated around a single component in the machine doesn't help matters any as far as wire routing complexity in the silicon. In this section, we introduce one alternative to the microarchitecture presented in the previous section that serves to reduce the complexity of the logic needed for operand searching of the reorder buffer at instruction dispatch time. We essentially will use the exact microarchitecture of the last section but with the addition of a new basic component. The schematic outline of this microarchitecture is shown in Figure 2.4.

The new component added to this microarchitecture is termed the *future file*. As with the reorder buffer, this component serves to hold speculative output register operands. However, unlike the reorder buffer, this component is not managed as a FIFO, or even as a content addressable memory array. Rather it is strictly accessed as a random access memory array. As we will see, this is the essential quality that will allow us to dramatically reduce the input operand acquisition logic that was required by the previous microarchitecture. The operation of the reorder buffer is unchanged with respect to its function of being updated by the outputs from the function units. It's function is also the same with respect to the transfer of speculative operands from it to the

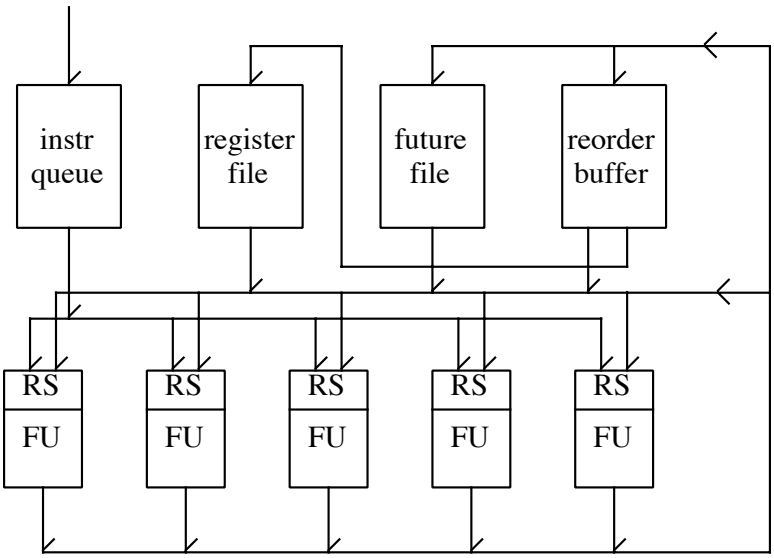

Figure 2.4: *Simplified schematic of a microarchitecture featuring reduced logic complexity for the reorder buffer component.* Shown is a very simplified schematic of the instruction queue, architected register file, future file, reorder buffer, and five of the reservation stations and function units of a microarchitecture that features reduced logic complexity for the reorder buffer. The introduction of the future file facilitated the simplified logic version of the reorder buffer.

architected register file during instruction commitment. However in this microarchitecture, when new output operands are created by the function units, they go to both the reorder buffer and the input of the reservation stations as before, but they also go to the future file. Both the update of the reorder buffer (as previously) and the future file is done using random access logic decoding. Now, unlike with the previous microarchitecture, during instruction dispatch, no content addressable search is needed for the reorder buffer in order to acquire the latest logical input operands. Rather a simple RAM-indexed lookup is performed on the future file to grab the needed register operands (if they are available). Like the previous microarchitectures, if the required latest input register operand value is not yet available (because it is going to be created by some function unit), a tag of the appropriate function unit is used instead.

Generally, the additional space used by the addition of the entire future file is not nearly as much space needed by the complex content addressable searching logic that was previously required at the reorder buffer. This microarchitectural arrangement shows both an example of an alternative to the regular reorder buffer arrangement as well as the seriousness of the problem of keeping the complexity of the logic needed for operand management functions to tractable levels. For our present purposes, this microarchitecture also shows how some creative arrangements of components (and even the introduction of a whole new component) can actually reduce the logic complexity

of the whole machine. Our own new microarchitectures (presented in subsequent chapters) follow somewhat on this theme by introducing a very different way to manage speculative operands. The second of our newly introduced microarchitectures effectively eliminates the problems of wire routing complexity in the silicon due to space concentration around a single component (the reorder buffer), as well as eliminate the need for prioritization logic for resolving which of several architected names should be acquired. We achieve this in a very novel way as compared with the microarchitectures presented in this chapter.

In this present microarchitecture (the most advanced that we have explored so far), we only had a single reservation station associated with each function unit. This limited the number of instructions that could be dispatched to the same number of function units within the machine. In the next section, as might be anticipated, we relax this restriction.

2.5 Function unit sharing and instruction issue flexibility

In this section we present some relatively minor enhancements to the microarchitecture of the previous section. Although the microarchitecture of the last section contains several function units, the number of decoded instructions that are in-flight within the execution core of the machine is only double the number of function units present within the machine. This was due to the fact that each function unit had only a single reservation station associated with it. In that arrangement, one instruction would occupy the reservation while another could occupy the function unit itself. However, in many instruction schedules created by either programmers or compilers, several instructions of the same instruction class many be grouped together in the static instruction schedule presented to the machine. This means for example that if there was a sequence of three instructions, in a certain section of the program, that all instruction dispatch would have to stall after the first two instructions were dispatched. This will be due to the structural machine hazard where by only two instructions of the same class can be in-flight at any given time. This sort of structural hazard is not particularly uncommon and can serve to reduce the amount of instruction level parallelism that can be exploited by the machine by keeping some function units idle while others are critically occupied. However, this problem is relatively easily mitigated through the addition on more than one reservation station associated with each function unit. This microarchitectural arrangement is shown in Figure 2.5. Note in the figure that we have shown two reservation stations associated with each function. Of course, now that we know the trick, we can extend this idea to have as many reservation stations associated with each function unit as we might like for different machine design points.

It should be noted that where there was only a single reservation station associated with each function unit, it operated in a first-in-first-out manner (although this represented a reduced limit

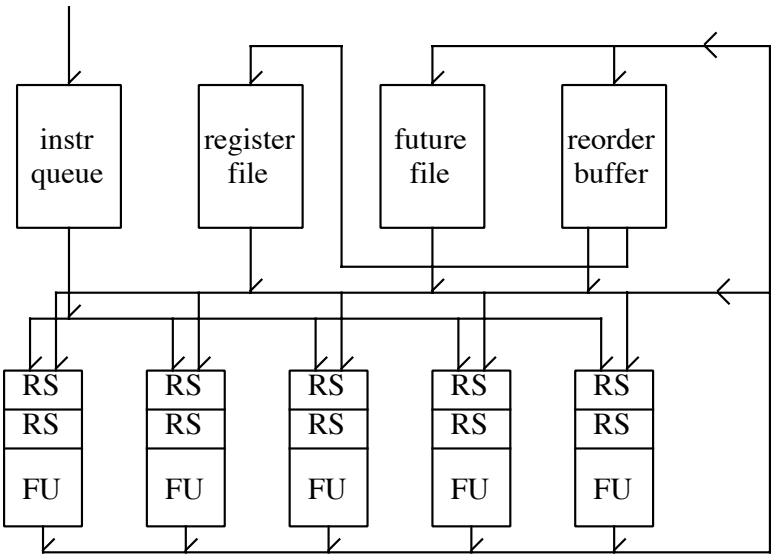

Figure 2.5: *Simplified schematic of a microarchitecture featuring multiple reservations associated with each function unit.* Shown is a very simplified schematic showing the instruction queue, architected register file, future file, reorder buffer, five function units, and ten associated reservation stations (two per function unit) of the processor execution core.

of the general idea). With the introduction of multiple reservation stations with each function unit, we have at least two choices about how to logically manage the flow of instructions through the reservations stations. One choice is to continue to manage all reservation stations in a FIFO order. More specifically, newly dispatched instructions enter the FIFO of reservation stations at the tail and instructions are only issued to function units from the head of the FIFO of reservation stations. This arrangement alleviates instruction dispatch stalling due to the unavailability of suitable reservation stations (instruction class structural hazard) while presenting only a slightly more complicated logic implementation for the reservation stations than the previous microarchitecture. With reduced instruction dispatch stalling, more instructions are able to become in-flight within the machine and more instruction level parallelism can be exposed within the static program schedule of the executing program. Further, there is increased likelihood that a greater percentage of function units will also now be busy (increasing overall IPC rate of the machine).

Another possible enhancement to the reservation station arrange (however something that is not visible in Figure 2.5 is that the reservation stations are not maintained in a strict FIFO order. Specifically this means that it is possible for any of the decoded instructions that may reside within any of the reservation stations associated with any function unit to be issued to the function unit when its input operands are all acquired (and assuming that the function unit is free). Previously, with the FIFO arrangement, only the decoded instruction at the head of the queue of reservation

stations was allowed to be issued to the function unit when ready. At first this relaxation of allowing one instruction to pass another in the overall order of execution might seem to present a problem for guaranteeing proper program state at commitment, but this is not the case. Since all instructions are still dispatched in original program order, they already are properly tagged so as to guarantee both correct input operands and commitment order. The passing of one instruction before another during instruction issue is no more a problem than the fact that one instruction of one instruction class was already allowed to pass another instruction of another instruction class through the presence of multiple function units in the first place. By allowing instructions queued to the same function unit to possibly pass each other during for instruction issue, overall execution performance is again somewhat increased because it means that the function unit does not have to remain idle under any circumstance where there is at least one of the instructions queued to it that is ready to execute due to acquiring all of its input operands. Previously, if the decoded instruction at the head of the queue had not yet acquired all of its input operands, the associated function unit would go idle once its present instruction was finished. Any time function units can be kept busy when they would have otherwise been idle, we have increased overall execution performance through increased IPC.

Another variation on this idea is that there could also be multiple function units of the same instruction class (more than just one) behind each of the sets of reservation stations serving that instruction class. We don't have a schematic of this arrangement but a more exaggerated example of this type of microarchitecture is instead presented in a subsequent section. From the discussion already, it is likely very apparent that a large number of different arrangements of the basic machine components that we have already presented is possible. Only some of the possible arrangements are presented in the remaining sections of this chapter. Our purpose is to provide a sufficient orientation to the microarchitectural possibilities so that the reader is well prepared for the consideration of our own microarchitectural introductions.

2.6 Superscalar and pipelined execution

In this section we briefly introduce the idea of superscalar instruction execution and the pipelining of the function units. For our purposes we can simply refer again to the microarchitecture shown in Figure 2.5 (as was also referred to in the last section). The idea of superscalar execution is that the machine can achieve an IPC rate of greater than one (1.0). This is accomplished through the combined effects (and requirements) of dispatching, issuing, executing, and committing more than a single instruction in any given clock period. Up until now, we have discussed the issuing, execution, and commitment of more than a single instruction at a time, but we had not specifically introduced the idea of dispatching more than one instruction in any given clock period. Although

the enhancement of dispatching more than a single instruction per clock could have been introduced in several of the previous microarchitectures, in historical practice it was not generally implemented until machine microarchitectures reached the richness and parallelism of roughly that presented in Figure 2.5. However, in more recent times, many microarchitectures of somewhat less parallelism than this current one now fairly routinely feature superscalar operation (at least theoretically so). It should be noted though that many superscalar machines may not average superscalar operation (achieving average IPCs of greater than 1.0) for many common integer sequential program codes. But many superscalar machines can still reach peak IPCs of approximately two for some program codes. But this is generally only achieved for machines that have their theoretical maximum IPC at something approximating four (4.0).

Although it may be obvious, the maximum IPC of a machine will never exceed the instruction parallelism at any point within the machine. Specifically, in order to achieve an IPC of greater than one (for example), more than a single instruction has to be fetched, dispatched, issued, executed, and committed on average in each clock of the machine. This is easier said than accomplished. Conflicts often arise within the machine as instructions progress through their various stages due to both true dependency hazards, structural hazards, control-flow-change mispredictions, and increasingly long waits for memory operands to come into the execution code from the memory hierarchy. Generally a fairly large amount of inherent possible parallelism has to be designed into a machine in order to reduce all of the causes for the many stalls that occur.

Another feature of many machines, along with superscalar operation, is that essentially all components of the microarchitecture are now generally substantially pipelined. Specifically, function units are now routinely pipelined and many of the function unit implementations of previously discussed microarchitectures above have used pipelined function units. Although the degree of pipelining in any machine is part of a set of design tradeoffs made for a particular design at the outset, the desire for faster clock speeds (shorter clock periods) has placed pressure on the need for deeper pipelines. Often the function units of a processor are among those that benefit most from pipelining. This is due to the fact that they often have a large amount of combinatorial logic organized as several levels of logic gates to carry out the whole of the execution of most instructions. For this reason, these units are usually among the most pipelined components within most machine designs. Of course, with the pipelining of function units, they are now free to accept newly issued instructions to them in each clock period. This design change alone serves to increase the overall execution performance of the processor.

In the following sections, we again pickup a discussion of variations on microarchitectures from which we built upon and extended for our own microarchitectures later on. In all subsequent microarchitectures, we will assume that they are capable of superscalar operation since that is a basic requirement for any microarchitecture that attempts to address program execution performance

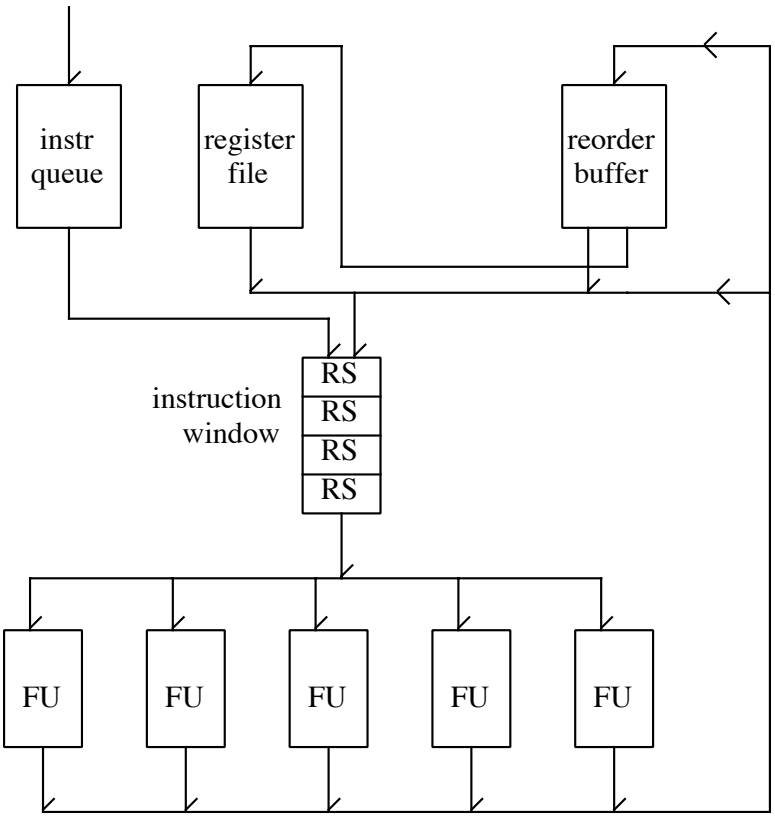

Figure 2.6: *Simplified schematic of a microarchitecture featuring an instruction window.* Shown is a very simplified schematic showing the instruction queue, architected register file, reorder buffer, five function units, and the instruction window (with four issue slots) of a microarchitecture featuring an instruction window.

(as with our new microarchitectures also).

2.7 The instruction window

In this section we briefly present a common variation of the last microarchitecture presented. In a previous section we alluded to the idea that more than a single function unit of the same class might be behind a set of reservation stations serving that instruction class. In this section, a variation of this idea is shown where more than a single function unit are all served by a common set of reservation stations. A schematic diagram of this type of microarchitecture is shown in Figure 2.6. In this microarchitectural arrangement, all of the reservation stations have been brought together into a single contiguous set serving all of the function units. This single set of reservation stations is shown in the center of Figure 2.6. Otherwise the operation of the remaining components is

essentially unchanged for the corresponding components presented in the previous sections. The single set of reservation stations that are grouped together is now termed the *instruction window* and now represents one of the key components of several current processor designs.

With this modification of grouping all of the reservation stations into a single, their utilization is increased because each of them can be used for any class of instruction (unlike the case when they are associated with specific function units). Their utilization is increased because all reservation stations are now statistically multiplexed rather than functionally multiplexed as before. But there is a price to pay for this improvement in reservation station utilization also. Now that all of the stations for the processor are grouped together, silicon wire routing congestion and complexity is substantially increased. When the entire instruction window is of small or modest size, the associated silicon routing problems are manageable. However, this arrangement suffers from not being physically scalable for large instructions windows (being approximately 256 for current designs). However, the benefits of having a statistically multiplexed pool of reservations stations is generally advantageous even in spite of the increased routing complexity of a centralized design. Variations on this basic microarchitecture (featuring the use of a centralized instruction window) has been adopted for use by several commercial processors, some of which include the Compaq Alpha family of processors [7, 80, 68].

One of our goals will be to realize a physically scalable machine and as might be expected, the centralized instruction window is not applicable for that particular goal. However, it will serve as a basis for the introduction of another arrangement for managing both the issuing of instructions (as the instruction window does) and the storage and ordering of speculative output operands (currently handled by the reorder buffer). A microarchitecture that builds on the idea of the centralized reservation station is presented in the next section.

2.8 The register update unit

In this section we present a microarchitecture that builds on the idea of the centralized instruction window and extends it in a way that will also manage speculative output operands. Effectively this microarchitecture will combine the centralized instruction window and the reorder buffer into a single machine component termed the *register update unit* (RUU). Each entry of the instruction window is essentially combined with one entry of the reorder buffer in a one-to-one pairing. The new combined entry in the RUU now serves both of the purposes of each of its prior two parts, but for the same dispatched instruction. A schematic of this microarchitectural arrangement in shown in Figure 2.7. The RUU is shown in the center of the figure and each entry within the RUU (each row) is shown in two parts. The reservation station part of an RUU entry is labeled RS (as were reservation stations in previous microarchitectures) and the reorder buffer part of an

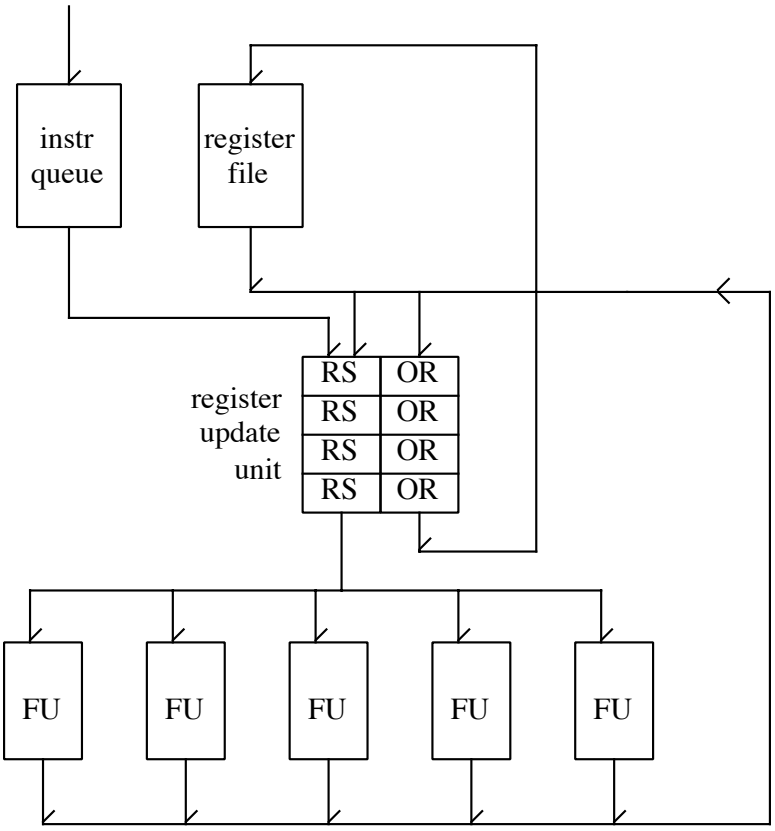

Figure 2.7: *Simplified schematic of a microarchitecture featuring a register update unit (RUU).* Shown is a very simplified microarchitecture schematic showing the instruction queue, architected register file, five function units, and a register update unit (consisting of four entries).

entry is labeled with OR (output register). This idea was first introduced by G.Sohi [132] and was originally meant to better facilitate the handling of precise interrupts in superscalar processors.

One feature of the RUU is that it also somewhat simplifies the instruction dispatch process and the required associated logic. With separate reservation stations and reorder buffer entries (residing in the reorder buffer), an instruction cannot be dispatched until both a reservation station and a reorder buffer entry is available. With a separate reorder buffer component (as in the previous microarchitectures), separate logic was used to manage the allocation of an entry from each of the two prior components. However when the separate component resources are combined as they are in the RUU, only a single allocation procedure and associated logic is needed in conjunction with the instruction dispatch process to manage the filling of instruction window entries. Note that the whole of the RUU now needs to be logically managed in a FIFO order since that was a requirement for the reorder buffer before it was integrated with the instruction window. We also note for completeness that the introduction of the RUU forces an equal number of instruction issue opportunities (through using the reservation station part of an RUU entry) and the number of speculative operand result storage opportunities. However, there has not been any significant evidence that this sort of a resource restriction is detrimental to overall machine performance.

We have presented this particular microarchitectural arrangement because we will build on this idea ourselves with our own new microarchitectures. However, rather than maintaining the idea of a centralized component (as the RUU was originally introduced), we will allow for the physical separation of individual entries (but generally still in groups consisting of some number of entries). Our adaptation of the RUU by allowing for entries to be physically separated will facilitate an overall physically scalable machine, which is one of our design goals.

2.9 General purpose execution pipelines

We impose upon our readers to indulge us in just one more variation on two of the microarchitectures presented previously. We do this due to the fact that one of the variations presented here served as the basis for one of our new microarchitectures. Each of the microarchitectures presented in Figures 2.6 and 2.7 can be modified to have unified execution units instead of the functionally separated function units as shown in the Figures. A unified execution unit can execute any of the instructions in the ISA of the machine rather than just a class of instructions as function units do. In particular, machines with a microarchitecture resembling that depicted in Figure 2.6 but with unified general purpose execution units rather than instruction classed function units is now very popular with many current microprocessor designs. Popular microprocessors that include the idea of a centralized instruction window along with general purpose function units include several Intel Pentium [100, 49, 89] processors and the DEC/Compaq Alpha family of processors [124, 7, 80, 68].

2.10 Disjoint Eager Execution

Although speculative execution has been employed in most present microarchitectures, these are all of a type where only a single speculative thread of control is speculatively pursued and handled at a time. However, speculatively pursuing more than a single program path simultaneously is also possible. Speculatively executing more than a single path within a program is termed *multipath execution*. In this section we briefly present three types of speculative execution with regard to how machine resources are allocated for speculatively executing instructions in each. These three types will be:

- single path execution
- eager execution
- disjoint eager execution

We will define and briefly explore each of these.

The type of control speculation where only a single speculative execution path is pursued and handled at a single time in the microarchitecture is termed *Single Path* (SP) speculative execution. Single Path speculative execution only considers allocating machine resources to a single speculative control flow path through the program being executed. The control flow path is not restricted to a single branch path [2] Rather, several branch paths may be speculatively executed. Branch paths, rather than basic blocks, are used throughout since it is only the conditional branches in the program that give rise to the ambiguity about which of the two output paths (or perhaps both) of a conditional branch should be considered next for resource allocation. As conditional branch instructions are speculatively encountered, they are predicted (either statically or dynamically) and only a single output path from the conditional branch is pursued for further speculative execution. The prediction of the conditional branch instruction itself is facilitated through any one of several mechanisms (branch predictors [127, 79, 171, 99, 93, 170, 14, 60, 117, 118, 26, 120]) that have been devised for that purpose. However, although many conditional branch instructions may be speculatively encountered, only a single speculative path of control flow is pursued and handled by the machine. Specifically, multiple speculative control-flow paths are not attempted. There is good reason for restricting speculation to a single control-flow program path and this is essentially related to the difficulty of maintaining state for more than a single simultaneous speculative path within the microarchitecture of the machine. The key microarchitectural component that was introduced (and commonly and generally employed) to facilitate the management of speculative machine state

[2] A branch path is the set of instructions from the first instruction following a conditional branch to the next conditional branch instruction and including it. It may contain any number of unconditional branch instructions within it. The key idea is that only conditional branches delineate the branch path.

for single path speculative execution is the reorder buffer. This was introduced by Smith and Pleszkun [128] and it offered an easy way to track and maintain speculative basic machine state (primarily the registers) when only a single speculative path is pursued.

However, beyond just speculatively executing along a single program path, pursuing two or more paths simultaneously (multipath execution) is also possible. We briefly discuss two possibilities for how to best pursue multipath in terms of how machine resources are allocated for the multiple speculative paths. How speculative machine state is maintained for multipath execution is another matter entirely and various methods (most of which are fairly complicated and painful for the computer designer) have been proposed. Rather our intent for the moment is to investigate how machine resources should be best allocated to the various speculative paths to be executed.

The second of the possibilities for multipath execution is that of speculatively pursuing both output paths of each conditional branch instruction as they are encountered in the instruction stream. This type of multipath execution is termed *Eager Execution* (EE) and is the simplest to perform in terms of deciding what output paths of a conditional branch instruction should be pursued – both are pursued making no restrictive decision necessary. The number of speculative execution paths multiply rather quickly with this form of multipath execution since each new conditional branch instruction that is encountered generates an additional speculative path once speculative execution proceeds past the current conditional branch instruction itself. The number of simultaneous speculative execution paths grows exponentially with respect to the number of conditional branches encountered – not the most favorable growth complexity a computer microarchitecture designer really likes to generally deal with. The rapid growth of simultaneous speculative execution paths constitutes a rather severe problem for Eager Execution and some mechanism has to be employed to mitigate the large resulting amount of execution path fanout possible.

Eager Execution has its roots in some early IBM machines. The first of these was the System 360 Model 91 computer. While not speculatively executing instructions from both output paths of conditional branches (it did not perform any speculative execution at all), it did fetch (or prefetch) instructions from both paths. [3] This idea of fetching from more than a single output instruction path following an unresolved conditional branch continued with the S/370 Model 165 and 168 machines. The introduction of the IBM 3033 machine [19] in the late 1970s, slightly expanded on the idea of speculatively fetching instructions by allowing for up to three paths to be fetched. This practice (fetching up to three instruction paths) was retained for the 3090 machine but actual multipath speculative execution did not yet occur. Although speculative execution did occur in the IBM 3090, only a single instruction path was speculatively executed. The IBM 3090 was the first machine to both speculatively execute instructions as well as prefetch down more than a single instruction path.

Wang and Uht explored more aggressive speculative handling of multiple conditional branch output paths employing the Eager Execution model using their Ideograph/Ideogram microarchitecture [166]. Wall studied the consequences of multipath Eager Execution by considering the amount of fanout allowed for at each conditional branch encountered. [162, 163] Due to the large number of simultaneous speculative paths that can be encountered, Wall used a technique where he limited the fanout growth by evaluating the *confidence* (the accuracy of a branch path prediction) that a particular branch path would be executed. Examples for attaining confidences for conditional branches has been explored by Jacobsen et al. [56] and others. The branch path confidences were calculated using static prediction probabilities acquired through program profiling. Heil and Smith explored a degenerate form of Eager Execution where only at most two paths are speculatively pursued at a time. [44] Tyson et al. [146] also explored dual path execution but did so in a new way where they evaluated branch path confidence and only splitting off an additional speculative execution path (for a total of two) for those branch paths with low confidence. The idea was that those branch paths that are the result of a low confidence conditional branch prediction should result in the spawning of both speculative output paths from the conditional branch. This makes sense since a high confidence in a conditional branch prediction means that there is not much to be gained through the spawning of speculative paths for both outputs of the conditional branch. This idea of Tyson et al. to only spawn additional speculative execution paths for those cases where they might be most useful in terms of possible instruction commitment is an important one and was actually introduced prior to their work. Finally, the idea of dual path execution was also recently explored by Aragon et al. [4] We next examine a generalized way to think about how machine resources should be applied to control-flow speculative execution when conditional branches are encountered.

It may now be intuitive that the best approach towards allocating machine resources to speculative execution paths is to make allocations to those speculative branch paths that are most likely to be committed. If we think about one extreme where all conditional branches could somehow be predicted with 100% accuracy (the oracle case), there would be no need for any parallel multiple speculative execution paths. In this case, any additional speculative paths beyond the single predicted path would only represent a waste of machine resources since there would be no likelihood of those additional paths ever being committed (commitment always happens on the paths resulting from the 100% accurately predicted conditional branches). This is the key idea embodied in the type of multipath execution termed *Disjoint Eager Execution* (DEE). This type of speculative multipath execution was first introduced by Uht and Singdagi. [148] They showed that this type of multipath execution was optimal in terms of machine resource allocation. In this type of speculative execution, machine resources are allocated for the speculative execution of the instructions within a branch path that is:

- not already allocated resources for speculative execution
- the next branch path among all possible that is most likely to be committed

The first of these criteria is already known within the microarchitecture, but how to determine the second of these criteria (the next branch path most likely to next commit) is the more interesting challenge.

A pictorial summary and brief comparison of the three forms of control-flow speculation that we have discussed is shown in Figure 2.8. In the figure, Single Path execution is shown at the left, Eager

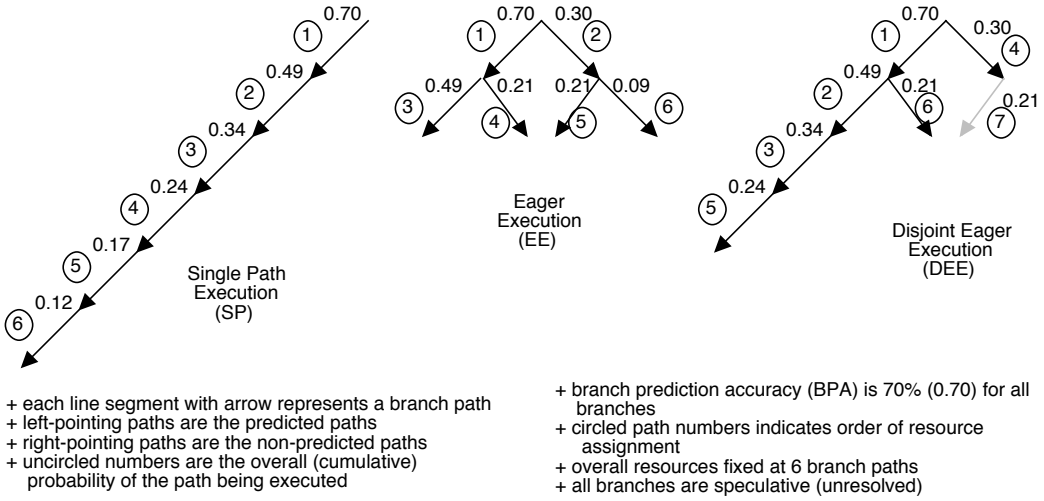

Figure 2.8: *Simplified diagram showing Single Path (SP) execution, Eager Execution (EE), and Disjoint Eager Execution (DEE).* Three approaches towards allocating machine resources for speculatively executing instructions is shown. Single Path (SP) execution is what is currently used in all existing microarchitectures. Eager Execution (EE) has been employed in a limited way for instruction prefetching, but Disjoint Eager Execution (DEE) provides the best model (of these presented) for appropriately allocating machine resources to additional simultaneous speculative execution paths.

Execution is shown in the middle, and Disjoint Eager Execution is shown on the right. Each type of execution is depicted with a series of lines, terminated with an arrow, where each line represents a branch path in the program code. At least six branch paths are shown for each execution type. The instruction located at the top of each branch-path set, as well as that instruction located at an arrow head represents a conditional branch. Each conditional branch instruction gives rise to two possible output branch paths. All of the instructions in all branch paths are speculative. The top of each set of branch paths represents the oldest outstanding speculative instruction, with the younger speculative instructions closer towards the bottom of each set of branch paths (in the directions of the arrows). Those branch paths proceeding down and to the left in the figure represent the predicted output path of the conditional branch instruction, while those branch paths proceeding

down and to the right (if any) represent the non-predicted output path of a branch.

Machine resources, such as reservation stations (issue stations within our own work) or execution function units, are allocated to the instructions within a branch path in the order shown by the circled numbers in the figure. The actual distinguishing feature of each type of execution shown is the order in which machine resources are sequentially allocated towards the speculative execution of future branch paths; it is not the same for each type of speculative execution. As can be seen in Figure 2.8, the case of Single Path execution only pursues a single speculative path (hence its name), while the other two types of execution (multipath execution) pursue multiple future speculative paths simultaneously. For the case of Eager Execution, it can be seen that future speculative branch paths are assigned machine resources in a very symmetrical way as each conditional branch is encountered; namely, for each output path of each branch. Since each output path of each conditional branch is speculatively pursued as it is encountered, this type therefore for its name as containing *eager* due to the eagerness of allocating machine resources for all branch outcome possibilities. And for the case of Disjoint Eager Execution, it can be seen that machine resources are not allocated in a symmetrical way around each conditional branch, hence its name *disjoint eager*.

For purposes of illustration in the figure, we have assumed that the branch prediction accuracy (BPA) is 70%. That is, seventy percent of the time the branch prediction is correct. Note that the predicted output path of a conditional branch instruction need not always be the same. Some branches may indeed exhibit alternating output behavior, but what we are concerned with is the prediction of the output path and not the actual path within the program code itself. History based branch predictors are very good at finding the predicted output path without regard for whether it is the same actual path or not (they generally maintain prior branch outcome history to help determine this). The branch prediction accuracy provides a sort of indication about the probability that a particular branch path will actually eventually be committed. When a branch predictor makes a prediction, this is a guess that the predicted output path will eventually be committed. The probability of the commitment is equal to the accuracy of the prediction, in the average case. As an aside, the per-instance case of prediction accuracy is generally termed *confidence*. The idea of branch prediction confidence can be thought of as the dynamic accuracy of a particular branch prediction, while the general accuracy is an average of all branch predictions. For the present discussion, we will ignore this distinction. Referring again to Figure 2.8, the numbers associated with each branch path that are not circled represent the cumulative probability of that branch path being eventually committed. The cumulative probability of any single branch path being committed is equal to the product of the cumulative commitment probability of its parent branch path with the likelihood of itself being the outcome of the immediate prior conditional branch. But the likelihood of a particular branch being the outcome of its immediate prior conditional branch

is just the accuracy of the branch predictor (if the current branch path is the predicted outcome) or one minus that probability (if the current branch path is the non-predicted outcome). If the branch prediction accuracy is p, then the likelihood of a leftward output path being speculatively executed, once the conditional branch itself is speculatively executed, is also p. While the likelihood of the rightward output path being speculatively executed is therefore $1.0 - p$. These cumulative probabilities can be seen in the Figure 2.8. Looking for example at the Single Path case (all leftward output paths – the simplest case), the cumulative probability of any single branch path being speculatively executed overall is just the BPA itself (in our present case being 70%) being raised to the power of the level (the root is at zero and the first branch path shown would be level one) of branch paths from the root of the speculative execution branching tree. Referring to the Single Path execution case, the first branch path at the top of the tree is labeled with a 0.70 cumulative probability, while the that of the next branch path is 0.49 (0.70 squared), et cetera.

With the calculation of cumulative branch path commitment probabilities for all three speculative execution types (as shown in the Figure with the uncircled numbers associated with each branch path), it becomes very clear as to how to best apply our goal of only assigning machine resources to the next branch path that has not already entered into speculative execution. One only has to (and should) allocate machine resources to branch paths in the order of increasing cumulative branch path commitment probabilities ! This is indeed the case for the Disjoint Eager Execution speculative execution type, and this can be seen upon examination of the cumulative branch path commitment probability as shown in Figure 2.8. Note (referring to the figure) that in the case of Disjoint Eager Execution, machine resources are allocated to the leftward output branch paths sequentially from the top of the tree until a possible level four is reached. At that point, the next branch path (and previously unallocated) with the highest cumulative commitment probability is not the next leftward output path from the branch instruction at the end of the third branch. That next leftward output path only has a cumulative commitment probability of 0.24 while the rightward (non-predicted) output branch path from the root of the tree has a cumulative commitment probability of 0.30 (being $1.0 - p$) in that case. This situation means that this rightward (a non-predicted path) branch path Any allocation of machine resources in an order different than increasing cumulative branch path commitment probabilities represents a suboptimal resource allocation (something of a waste of machine resources). As the oldest conditional branch of the speculative execution tree commits, any incorrect speculative branch paths that sprouted out from that point in the tree (if any) are abandoned. We also here introduce two terms that are frequently used when discussing Disjoint Eager Execution. The speculative execution path that always follows the predicted output path of a conditional branch is termed the *main-line* (NL) path. There is only a single speculative main-line path possible since there is only a single path that can be derived from all speculative conditional branches being predicted taken. All other possible speculative paths are

termed *disjoint eagerly executed* (DEE) paths.

Finally, Disjoint Eager Execution can be thought of as being *iso-probability speculative wavefront execution*. We can think of all of the speculative execution within the machine as having a wavefront, where the wavefront is the boundary between those instructions being currently speculatively executed and those that are possible feasible speculative instructions (not yet being speculatively executed). However, since all instructions within a branch path have an equal probability of being committed (ignoring exceptions and interrupts), the granularity of the speculative wave advance actually occurs with branch path increments. What is desirable is to allocate free machine resources to those branch paths in front of (outside of) the wave that have the highest probability of commitment. This is just the cumulative branch path commitment probability that we have introduced. If this mechanism is followed, the wavefront approximates a locus of points of iso-probability (same probability) commitment where each point is represented by a speculative branch path. Since the total resources available for allocation within any given machine is fixed. The area within the wavefront essentially (or generally) remains constant (equal to the total resources of the machine) and the wavefront expands as instruction (branch path) commitment occurs at the trailing edge of the wave.

2.10.1 Use of average branch prediction accuracies

In our example so far, we have used the branch prediction accuracy as something of a substitute for branch prediction confidence. Branch prediction confidence is the dynamic probability that a particular branch prediction is correct. The use of the term *dynamic* for branch prediction confidence gives rise to the use of the term *static* to describe using the branch prediction accuracy for commitment probability calculations. Although a more accurate way to calculate the probability of a particular (dynamic instance of a) branch path being committed would be to take the product of all prior branch path confidences, this is difficult and resource intensive in terms of actual machine hardware. Think about the idea of storing and multiplying branch prediction outcome confidences in order to calculate the cumulative commitment probability of each branch path to get an idea of the unpleasantness of that approach. Happily for the computer designer, research by Uht [152] has shown that the use of the overall branch prediction accuracy (BPA) is sufficient, and almost as good as, to approximate commitment probabilities as compared with using individual dynamic branch prediction confidences. Note that although branch prediction confidences need to be determined dynamically during execution, that is not necessarily the case for the branch prediction accuracy (BPA). This is because BPA can be determined as an average over many program executions using a particular branch predictor design. The use of constant predictor confidences (otherwise just the accuracy of the branch predictor) is termed the *static tree heuristic* and was first coined by

Uht [148].

2.10.2 Implications for large instruction windowed machines

Let us take a moment and consider the applicability of Disjoint Eager Execution from proposed machines (no real machine today uses this approach). Let us start by assuming a more general expectation for branch predictor accuracy with the latest predictors. Many branch predictors have shown branch prediction accuracies (BPAs) in the high 90 percentiles but for our present purposes let us assume that we use a somewhat conservative branch predictor with a BPA of 90%. So the commitment probability p of the predicted output path of the first conditional branch encountered would be 90%. Let us also assume that an average branch path is nine (9) [3] Recall that main-line path of the speculative execution tree using Disjoint Eager Execution is that path where all constituent branch paths were chosen as being the predicted output paths of all encountered conditional branches. Well, the situation of speculatively executing only the main-line path is also exactly the equivalent of the Single Path case and is what is done in most current microarchitectures. Question: how many instructions should be speculatively executed along the main-line path before a DEE path should be spawned ? We already have our two assumptions needed to answer this question. With a BPA of 90% and an average number of instructions of nine making up each branch path, we first need to find the number of branch paths along the main-line path until the cumulative probability for commitment of the next main-line branch path would be less than 10% ($1 - p$ or $10.0 - 0.9 = 0.1$). Using the equation:

$$h_{ML} = \text{floor}(\ln(1.0 - p)/\ln(p)) \tag{2.1}$$

where:

- p is the branch prediction probability (BPA)
- h_ML is the height of cumulative commitment probability for a
- floor is the mathematical function for getting the integer floor of a number
- ln is the natural log function

we can calculate the height in branch paths of the main-line path before machine resources should be allocated to the first DEE path. Having calculated h_ML, multiplying this by the average dynamic instructions per branch path gives the total number of instructions that should be speculatively

[3] The actual dynamic branch path instruction frequency for the ten benchmarks BZIP2, CRAFTY, EON, GCC, GZIP, PARSER, PERLBMK, TWOLF, VORTEX, VPR, from the SpecInt-2000 suite when executed on an Alpha ISA is 9.2 when executing 100M instructions after skipping 500M instructions.

executed before resources are allocated for a DEE path. In our present example, our $h_M L$ turned out to be 21 and with nine instructions per branch path, our total ML path instructions are (or should be) 189. What is interesting with this example is that only recently have machines started to accommodate upwards of 200 outstanding speculative instructions. This result shows that for the Alpha ISA (as used in our example) and for a set of SpecInt-2000 benchmarks, that any machine that does not accommodate over 189 speculative instructions should not implement Disjoint Eager Execution. Interestingly this has been the case for actual commercial machines to date. This analysis places something of a lower bound on the size of a machine, in terms of how many instructions it should have speculatively in-flight, before it should consider using Disjoint Eager Execution.

Ahuja et al [1] studied limits of performance speedups when using multipath execution, but they only explored relatively modest sized machines (consistent with machines of today of at or under 200 outstanding speculative instructions). They showed some performance benefits of multipath execution almost in light of our analysis above about when to spawn a first DEE path, their results are either somewhat optimistic or likely not worth the additional hardware resources and complexity for implementing an actual machine capable of handling multipath execution.

Note that our assumptions about instructions per branch path and branch prediction accuracy may be somewhat conservative. For example, programs other than sequential integer ones, featuring floating point oriented scientific applications for example, may easily have more instructions per branch path than nine or ten. One reason for this is the greater degree of software looping unrolling employed for these sorts of work loads. Further, scientific work loads can often have a higher average branch prediction accuracy than integer sequential programs. For both of these reasons, the number of instructions that should be executed on the main-line path before spawning a DEE path could be much greater than 200 or 300. This makes the employment for Disjoint Eager Execution much less attractive for conventional and contemporary resource sized machines. For these reasons, it is very likely that only future larger sized machines that can hold several hundred instructions in-flight should even consider exploring the use of Disjoint Eager Execution.

There have been some machines (in addition to those already discussed previously) that have explored the use of multipath execution (not DEE) without having an instruction window size of approximately 180 or higher. We look at some of these proposed machines next.

2.10.3 Additional proposed multipath machines

Besides the dual-path machines of Heil et al [44] and Tyson et al [146] already discussed above, other machines have been proposed exploring multipath execution in slightly more aggressive ways. One such proposal is the PolyPath microarchitecture by Klauser et al. [74] Their machine implemented

a limited form of multipath Eager Execution. They used neither the Disjoint Eager Execution model, nor the static branch tree heuristic to make alternative path spawning decisions. Rather than used additional hardware to maintain branch predictor confidences and to make spawning decisions from those. However their approach limited how many spawn opportunities they could handle. Another whole area of a sort of multipath execution has been explored by Tsai and Yew with their Superthreaded Architecture [142] and others such as Wallace et al. [164] Wallace (and others since them) explored using *simultaneous multithreading* (SMT) to spawn additional program threads when certain events indicate that a possible performance speedup might be available. This work built on the original work on SMT by Tullsen et al. [145] Additional SMT threads are more akin to the idea of subordinate multithreading and are not as closely related to multipath execution as we have discussed so far. But we mention this work due to its tangential relationship to the multipath execution that we are concerned with. Generally, the various subordinate threading approaches are used as a higher granularity of parallelism than what strict multipath execution does.

2.10.4 Our use of Disjoint Eager Execution

We will be introducing two microarchitectures based on the Resource Flow execution model. The first of these (to be called OpTiFlow) explores resource flow execution primarily in the context of conventionally sized microarchitectures. These would be microarchitectures of having approximately 256 or less instruction outstanding at a time. For this microarchitecture we do not apply the technique of Disjoint Eager Execution since the benefits are likely to be small – and certainly small compared with the additional hardware logic to implement it. However, we will also be introducing a scalable microarchitecture, also based on resource flow execution, but this machine is oriented towards having very large instruction windows of 256 or more outstanding instructions. For this latter microarchitecture, we will use the ideas of Disjoint Eager Execution and also the static branch tree heuristic (using branch prediction accuracies for guiding the spawning of DEE paths). This latter microarchitecture is discussed in more detail in Chapter 9.

2.11 Other recently proposed microarchitectures for ILP extraction

In this section we briefly discuss some of the more recently proposed microarchitectures oriented towards extracting a large amount of ILP as compared with the commercially available machines marketed to date. These microarchitectures actually come in two basic types. The first type of these microarchitectures are entirely new in that the require a completely new ISA than any existing

machine ISA. These microarchitectures cannot natively execute any existing legacy program codes and are therefore quite unlikely to be implemented by any general purpose processor manufacturer for some time to come (although it isn't entirely impossible that some new general purpose processor ISA might be introduced someday – within the next 50 years for example !). For this reason, these types of microarchitectures are not likely to be ever implemented any time soon. Although these microarchitectures are not strictly within the problem domain that we are addressing with our work, they do introduce some concepts that are noteworthy for background consideration. The second type of microarchitectures discussed can be implemented for any ISA, including legacy ones, and so these serve as somewhat better models for inspiration of new ideas than those that require entirely new ISAs. For many new microarchitectural proposals, the requirement to be able to execute legacy programs codes (implement a legacy ISA) is often a nontrivial one and so they better serve as both a starting point and a comparison point for our own microarchitectural work.

Microarchitectures that have employed the use of multiple execution units are the Multiscalar-like processors [133, 136], the Superthreaded processor model [142], and the Parallel Execution Window processor model [66]. Other microarchitecture proposals such as the MultiCluster machine model by Farkas et al. [30] are also in this category. In particular, the Multiscalar processors have realized substantial IPC speedups over conventional superscalar processors, but they rely on compiler participation in their approach.

Another attempt at realizing high IPC was done by Lipasti and Shen on their Superspeculative architecture [85]. They achieved an IPC of about seven with conventional hardware assumptions but also by using value prediction. The use of value prediction was introduced with these microarchitectures and represents the first attempt at breaking the data dependency chains in the target programs to be executed. Although studies of this technique (such as by Gonzalez and Gonzalez [39]) and these microarchitectures initially showed substantial promise as a way to achieve considerably improved ILP extraction on integer sequential program codes. More recent expectations for value prediction have not been as optimistic as originally anticipated. However, the ability for a microarchitecture to accommodate and manage the particulars of allowing for the application of value prediction dynamically during execution may still be a desirable goal; even more so if it can be achieved as part of or in a similar way to how control flow prediction is managed. Our own newly introduced microarchitecture does exactly this, managing speculative instruction execution due to both control-flow prediction and value prediction in a similar way. As will be seen, our new microarchitecture also provides for a more extensive use of speculative execution in a very generalized way. Our generalized management of speculative execution also eliminates some of the awkwardness of the previous proposals using value prediction.

Nagarajan proposed a *Grid Architecture* of ALUs connected by an operand network [97]. This has some similarities to our work. However, unlike our work, their microarchitecture also relies on

the coordinated use of the compiler along with a new ISA to obtain higher IPCs.

As we will propose in Chapter 7, we explore the use of predication, and specifically a new way to perform dynamic predication within the hardware at execution time, for the purposes of eliminating the deleterious effects of conditional branches as well as possibly exploiting the benefits of speculatively executing control independent code beyond the join of simple-construct branches (like simple single-sided hammock branches for example). Proposed machines that have explored the possibility of handling simple branch constructs and subsequent control independent instruction execution have been introduced by Rotenberg and Smith [110], Klauser et al. [72] and the Skipper microarchitecture by Cher and Vijaykumar. [15] The Rotenberg machine itself is based on the previous Trace Processor microarchitecture by the same author. [111]

Finally, we want to make note of the Ultrascalar processor microarchitecture introduced by Henry et al. [47, 46]. This microarchitecture shares some similarities to those that we will introduce and also the goal of extracting larger amounts of ILP from difficult program codes. Specifically, this microarchitecture adopts something of a unified instruction window and reorder buffer (similar to a register update unit), but they further integrate a general purpose execution unit with entry of the structure. The result is something like an instruction execution station for each entry of the entire structure. This microarchitecture also achieves what the authors term asymptotic scalability. This is somewhat aptly named since they adopt an operand passing interconnection network for transferring operands amongst execution stations and which grows as $O(log(n))$ with respect to the number of execution stations employed in any particular design. For our own work, we consider their interconnection network approach too unwieldly for the amount of machine scalability that we are targeting (very large), and also retain a sharing approach with the execution units since they are often some of the largest logic blocks in current fast machines. Specifically, we feel that allocating a full general purpose execution unit for each instruction execution station is not a good use of silicon space when speculative execution is also employed (requiring possible re-execution). Further, in the Ultrascalar processor, when instructions must wait for input operand acquisition, their corresponding execution units are idle, representing a substantial loss of silicon utility.

In the subsequent sections, we briefly further explore some of these other proposed machines for extracting ILP from sequential programs.

2.11.1 Multiscalar microarchitecture

Much attention has been paid to the Multiscalar [133, 136] microarchitecture as a proposal for extracting higher ILP from programs, so we will discuss it a bit further also. This microarchitecture was an attempt to exploit instruction level parallelism at a course grain within programs. This is not a conventional multithreaded approach where threads are spawned either by the programmer

or by the compiler. Rather this approach attempts to take a single threaded program and compile it in a way that prepares it for execution on a machine that will execute multiple parts of the program simultaneously.

The process is largely compiler driven, and performance of the resulting execution is largely dependent on the ability of the compiler to segment the original programs into what are termed *tasks* for execution by the underlying hardware. The compiler attempts to identify a section of the original program that will make up a certain unit of computation that is as independent (both control and data) from subsequent units of code (also turned into tasks) as possible. Good candidates for tasks might be the entire execution of a loop, for example, possibly including its subroutine calls. Another candidate for becoming a task might be a single subroutine call. All identified tasks are control-dependent on the previous tasks. There are some obvious limits on how well a compiler can identify units or code that should become execution tasks, and this was one of the main limitations of this approach towards parallel execution.

Once the compiler identifies a section of code that should become a task, it compiles it into what is termed a *task block* that will go into the output binary file. The task block consists of both the program instructions used to execute the unit of code that was identified by the compiler as well as additional information in the form of a *task descriptor* that contains information about what architected machine registers the task requires from previous tasks that it must wait for, as well as those architected registers that the present task must output and forward to subsequent tasks. The compiler identifies which machine registers are possibly needed as inputs to the present task and constructs a *accum* register mask containing them. The compiler also generates all possible output registers from the current task and places them into a *create* register mask. Both masks are placed into the executable binary file within the task descriptor of the current task. Since both the precise input machine registers needed and the output registers that are generated can only be determined through dynamic execution, the most conservative calculation of each type is made by the compiler (through its static analysis of the program code). Once the entire program is divided into tasks, the task blocks are placed into an output binary file in a way that also indicates their sequential control flow order. The idea of creating task blocks and placing them into the binary executable file makes this Multiscalar machine not binary compatible with any existing machine, nor can it be. Although the machine instructions within each task block resemble instructions of other machines (and could indeed be the ISA of some other machine), the additional task descriptor information also within each task blocks makes this architecture fundamentally different and new compared with any existing machine. Once a binary executable object file is created, it is ready to be loaded into the machine and executed on its hardware. The machine hardware is then tasked with attempting to execute multiple sequential tasks in parallel on substantially separate execution processors.

The underlying hardware consists of a ring of processing units (PUs) that are simular to conventional processors. This ring structure [104] is common to most all microarchitectures and even a degenerate form of it can be seen in the original Tomasulo idea of using reservation stations and its Common Data Bus to loop output registers around to both the register file and to be used as subsequent inputs to subsequent instructions. Each processing unit includes its own instruction cache, a *processing element* (for executing instructions), and a machine register file. Interesting, these interconnected processing units more closely resemble a set of independent but connected processors making up a multiprocessor. However in the Multiscalar microarchitecture all of these processing units are dedicated towards the execution of a single program (after it is broken up into tasks by the compiler). This arrangement is shown in Figure 2.9. At the top of Figure 2.9 is

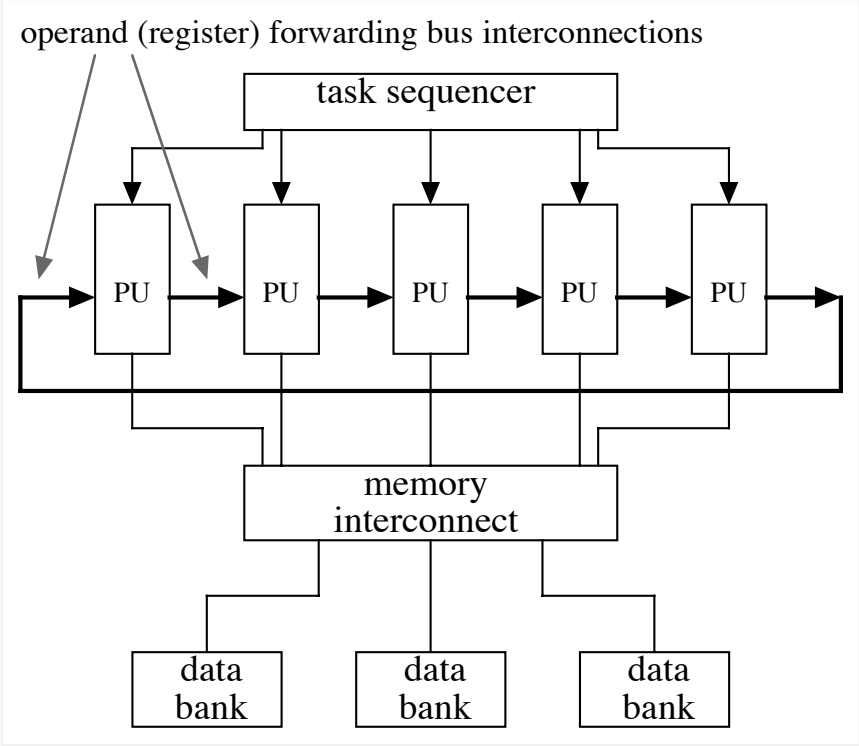

Figure 2.9: *A simplified Multiscalar microarchitecture.* The Multiscalar microarchitecture consists of a ring of processing units (labeled PU), each of which is used to execute a control-flow sequential task of a single program. Processing units forward output register operands to subsequent processing units through a forwarding bus.

shown the *instruction sequencer*. This component fetches tasks blocks from memory (having been loaded from the binary executable). It then reads the task descriptors from each task block and arranges for both the instructions of the task and the *accum* and *create* register masks, along with

additional information from the task descriptor, to be sent to a particular processing unit as one becomes free.

Also shown are the set of processing units (PUs) interconnected in a ring arrangement (five PUs are shown in the figure). Since each processing unit contains its own instruction cache and fetch unit, fetch and decode occurs across all processing units in parallel. However, since task blocks were first read from memory using the instruction sequencer component, there is still a centralized point in the machine where all instructions flow through. The ring interconnection allows for output register operands from one processing unit (executing a single task) to be forwarded to the next sequential processing unit. The number of processing units for any given Multiscalar machine is a design point decision. Memory operands are handled differently and are discussed below. Although the set of processing units form a ring structure based on their hardware interconnection, a head and tail pointer are also maintained within the machine to point to the first processing unit that is executing a task (the head) and the last that is executing a task (the tail). If all processing units are occupied (which does not necessarily always occur due to processing unit flushes), the head and tail processing units will be physically next to each other, with no free processing units available. Since the head processing unit contains the oldest assigned task, it will commit first. All commitment among processing units occurs in order. When a commitment of a processing unit occurs, the head pointer is advanced to the next processing unit. Commitment always occurs from the head of the ring and proceeds in-order to subsequent processing units. Likewise, new tasks are assigned to the free processing unit that is adjacent to and downstream from that processing unit currently designated as the tail. When a new task is assigned to a free processing unit, the tail pointer is advanced to point to that newly assigned unit.

Unlike register operands, memory operands being either read or written from the processing units got to a memory interconnection component. This component then switches the memory access to one of several *data blocks* within the machine (shown at the bottom of Figure 2.9). Each data block itself is made up of two subcomponents (not shown in the figure). The first of these is the famous Address Resolution Buffer (ARB) [34]. The second block is a data cache. The ARB serves to disambiguate memory accesses from the several processing units and enforces correct memory ordering. If an ordering violation is determined to have occurred, a single is sent to stop execution in the processing unit following (in ring order) the one that it has a memory conflict with and the execution of its tasks and those of all subsequent processing units is stopped and squashed. This is sort of equivalent to a pipeline in a convention pipelined processor, except that now entire processors with their executing tasks need to be flushed. Obviously, for performance reasons, it is hoped that this event doesn't occur often.

Since the individual processors are essentially similar to almost an entirely conventional independent processor, any type of these existing microarchitectures might be employed for the

construction of a Multiscalar machine. This is something of a design advantage of the Multiscalar approach since different processors of varying complexity can be used in any single Multiscalar design so as to provide different performance design points. For example, processing units could consist of simple processors with 1-way issue and little or no instruction parallelism to complicated superscalars with abundant speculative execution and execution units. Also, since each processing unit operates substantially independently from the others, the tasks executing in each look more like individual threads than multiple paths of the same program. In fact, amazingly, each task within each processing unit has its own program stack. This allows each task to execute subroutine calls and returns independently from the other tasks (even though all tasks are executing the same program thread). So the idea of a program (even s single threaded program from the perspective of the programmer) having a single stack is blown away. This maintenance of multiple program stacks within what was really a single threaded program shows how different this execution is from either multipath execution (a single program stack) or multithreaded execution (multiple program stacks). As might be imagined, this use of multiple program stacks substantially complicates machine control.

With regard to performance, this machine microarchitecture (and new architecture) achieves program IPC averages of about 2.9 over integer sequential program codes. This is achieved using eight out-of-order superscalar processors, each having 2-way issue capability, as eight processing units.

Ideas from this microarchitecture that are related to our own new work includes the basic ring arrangement for parallel instruction processing and operand forwarding. Also, Multiscalar represents *control dependence decentralization* [104] and our new microarchitectures also feature a control dependent decentralized arrangement. Essentially the rest of this microarchitecture (Multiscalar) is very different from our own microarchitectural introductions. More specifically, this microarchitecture features binary incompatibility with all existing ISAs as well as being very much reliant on its compiler for splitting the original program into task blocks. These are among its more conspicuous departures from our own work. In addition to being binary incompatible with any existing ISAs, Multiscalar is also very much dependent on the ability of its compiler to prepare and organize the target program for execution on the machine. One important task of the Multiscalar compiler is its very nontrivial ability to employ very sophisticated techniques to partition the target program into potentially parallel executing task blocks. Non of this is required for our own microarchitectures. Rather, entirely conventional compilers can be employed, including all existing ones for a given ISA, when compiling for our microarchitectures. This is possible since our newly introduced microarchitectures are able to implement any ISA (either a new one or legacy ones).

2.11.2 Parallel Execution Windows

The Parallel Execution Windows (PEWs) processor model [66] is another attempt at decentralizing program execution for the purpose of extracting higher ILP from sequential program codes. This machine is really an attempt at implementing something of a data-flow machine since it starts by determining program dependencies in order to separate code blocks out to different execution resources based on relative program dependencies. This machine employs a set of *execution windows* (EW), hence its name, where each consists of an independent window of instructions that gets assigned to it by centralized assigning hardware. Further, once each execution window is assigned instructions, they can operate somewhat independently from each other (therein providing ILP performance speedups). Unlike some other approaches to instruction parallelism, no new ISAs need be invented to make use of this machine proposal. The microarchitecture of the machine reads and analyzes instructions for possible ILP extraction without the assistance of any compiler.

The centralized instruction assigning hardware decodes and analyzes the instruction stream and attempts to break it up into multiple groups of instructions that are largely data-dependent on each other within the same group. Once data-dependent groups are identified, they are assigned to an execution window. The basic philosophy of this machine is that by placing groups of instructions that form a highly data-dependent chain within the same execution window, communication among different execution windows (in the form of register value transfers) will be minimized. This machine idea represents an attempt at *data dependence decentralization* [104]

The PEWs microarchitecture is shown in abbreviated form in Figure 2.10. At the top of Figure

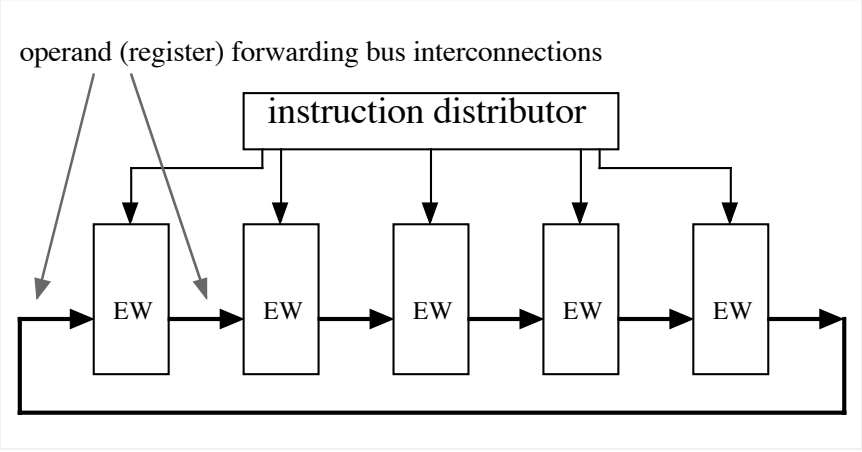

Figure 2.10: *The Parallel Execution Windows (PEWs) microarchitecture.* The Multiscalar microarchitecture consists of a ring of processing units (labeled PU), each of which is used to execute a control-flow sequential task of a single program. Processing units forward output register operands to subsequent processing units through a forwarding bus.

2.10 is shown the *instruction distributor*. This is the name the PEWs designers coined for the unit that performs the data dependency analysis on the program instruction stream and then assigns a data-dependent group of instructions to an individual execution window of the machine (labeled EW in the figure). The set of execution windows (five in the present example) are shown horizontally in the center of the figure. They are interconnected with a set of unidirectional operand forwarding buses into a ring arrangement. Outputs from execution windows are forwarded to other execution windows that are waiting for them using the forwarding buses. Although an output from any single EW may have to be forwarded to a non-adjacent EW, it is first forwarded through as many intermediary EWs as necessary. The operand forwarding process (not shown in the figure) uses a mechanism termed a *forwarder*. Control-speculative execution (using branch prediction) is allowed within individual execution windows and recovery from a misprediction is likewise performed in parallel. Although this PEWs microarchitecture may seem to closely resemble that of Multiscalar at first look (and the PEWs creators point this out also), it differs fundamentally in the following respects:

- PEWs is entirely binary compatible with existing ISAs where Multiscalar is not
- PEWs perform data dependency analysis on the program instruction stream for EW assignment while Multiscalar only has to assign tasks to its PUs
- the arrangement of EWs in the PEWs microarchitecture does not represent a strict control-flow ordering as the PUs in Multiscalar do
- the EWs in the PEWs microarchitecture are generally not nearly as sophisticated as PUs can be in Multiscalar

Simulated performance for this machine proposal has yielded program IPCs of between 2 and 4 for the largest machines investigated. This is substantial when one considers that most current superscalars still only average an IPC of around 1.0 or less.

This machine was clearly an attempt at exploiting the benefits of minimizing the communication overhead among decentralized computing resources. This was achieved by taking a data-flow oriented approach. Our proposed microarchitectural approach does not attempt to perform either a data flow analysis of instructions nor to even determine data dependencies before instruction execution. So in some sense our work is almost entirely opposite of the PEWs approach. However, both approaches do share the desire to decentralize execution and to deal with the management of forwarding output operands to other separated instructions or code blocks that requite those as inputs. Additional, our work will also employ the idea of forwarding operands (although in a somewhat different way), and using components somewhat resembling the forwarder mechanism

employed in the PEWs microarchitecture. We will call our forwarder hardware *forwarding units* and they will have much more capability than was introduced with PEWs. Just as this microarchitecture can implement any ISA (new or legacy), likewise that will be a feature of our own microarchitectural work.

2.11.3 Superspeculative microarchitecture

Unlike the previous microarchitectures that we have just examined, the Superspeculative microarchitecture by Lipasti and Shen [85] is actually much more closely related to a conventional superscalar processor than anything else. It is basically a superscalar processor using reservation stations with some added hardware for making a variety (a large variety) of value predictions with. Like many conventional superscalar machines, the Superspeculative microarchitecture is binary compatible with all existing ISAs (no special compiler assistance is needed). A diagram of the basic structure of this microarchitecture is shown in Figure 2.11. This is essentially identical to

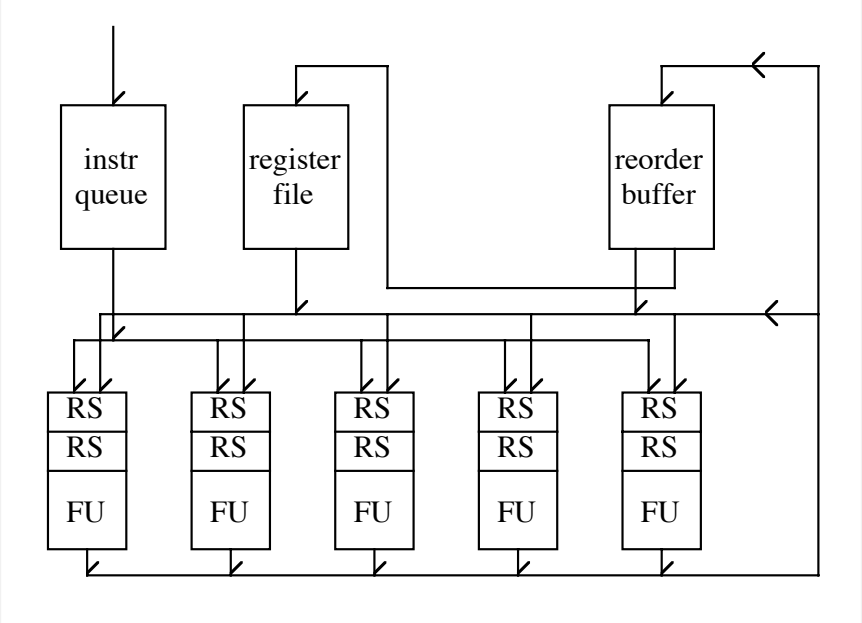

Figure 2.11: *The Superspeculative Microarchitecture.* The Superspeculative microarchitecture is essentially the same as a conventional superscalar using reservation stations but it differs from existing machines in that it uses value prediction.

a conventional superscalar that we introduced previously and which is shown in Figure 2.5. In fact it is almost identical except that this microarchitecture employs two elements that we had not previously introduced. These are:

- the use of a trace cache in addition to a conventional instruction cache
- the use of value predictors for making several predictions within the microarchitecture

Like most conventional superscalar processors of today the Superspeculative microarchitecture also makes use of branch predictors for break control dependencies. The use of the trace cache is not new and it was first introduced by Intel and subsequently further explored by others. [111, 110] The trace cache basically provides an instruction fetch enhancement over a conventional instruction cache, so it is not a fundamentally new way of handling machine execution in that sense. Also like conventional and proposed machines, the Superspeculative microarchitecture employs a centralized instruction fetch and decode strategy. The real novelty of this microarchitecture is its use of value prediction. The addition of value prediction along with existing control prediction makes this microarchitecture perform highly speculative execution, and hence its name *Superspeculative*. The concept of value prediction was first introduced by Lipasti and Shen in 1996 [82, 87] and they continued to explore it subsequently. [81, 84, 86] Many others, including Sazeides and Smith [117, 118], Gonzalez and Gonzalez [42, 40], Calder et al. [12], Rychlik [113], Thomas [138], and others have also explored the benefits of value prediction.

The basic idea of value prediction is to predict a value in a similar way as a conditional branch might be predicted, except that in the case of a conditional branch the output is only a boolean value (being either TRUE or FALSE) while in the case of general value prediction, the output is a value what consists of a number of bits (not uncommonly the width of a word in the machine architecture). In a certain sense then, even branch prediction is itself a form of value prediction, except that it use is limited to predicting the binary outcome of a conditional branch instruction. Once a value is predicted and it is speculatively used in the execution of an instruction it must eventually be verified to be correct by checking it against the correctly known resolved value (the value to be committed to the permanent program state). Like with branch prediction, if a value prediction is correct, no additional action is required and generally a performance speedup has been achieved. However in those cases where the predicted value turns out to be incorrect, corrective action needs to be taken within the microarchitecture to rollback execution and to somehow (there are different means available) re-execution the offending instruction. An obvious use for value prediction is to predict the input operands of an instruction before those operands have been computed by previous instructions that produce them. Most work has indeed focused on this use of value prediction. However, although many have explored value prediction for predicting instruction input operands, Lipasti and Shen (the originators of the whole idea) have taken this a couple of steps further in the Superspeculative microarchitecture. In additional to just predicting values for operand data (be it register or memory), microarchitecture also predicts instruction dependencies themselves, as well as memory aliases.

In the case of operand value prediction, the Superspeculative microarchitecture predicts the source operand values that are inputs to instructions. In other microarchitectural proposals, the outputs of instructions are predicted and used as inputs to subsequent instructions. Obviously, there is a large similarity between source value and output value prediction since both effectively predict values between the execution of instructions. The the choice of which is used is largely a design convention of a particular microarchitecture. Predicting instruction dependencies is useful because the logic required to calculate instruction dependencies is substantial. If instruction dependencies can be predicted, instructions can be dispatched faster (without waiting for what it is dependent upon). Likewise, memory alias prediction speeds up those memory accesses that do not eventually result in a disambiguation conflict. Of course, all prediction requires something like squash or roll-back when the predictions are incorrect, but there are enough times when a prediction is correct to make it often worth the additional hardware mechanisms to provide it. Most research has shown that with adequate value predictor hardware, an average prediction accuracy of approximately 50% is achieved. This is far below the typical average accuracy rate for conditional branches (typically around 90% or better), but bear in mind that the outcome of a generalized value prediction is usually a whole machine word (for example 32-bits or 64-bits) rather than just a single boolean value, as the case is for conditional branch outcome prediction.

To summarize, the Superspeculative microarchitecture makes the following types of predictions:

- conditional branch
- instruction dependencies
- memory aliases
- operand values

This is much more prediction than any current or proposed machine makes, but the simulated results are impressive. The Superspeculative microarchitecture has achieved IPCs of about 7.3 on average across the Spec-95 benchmark suite. That is an incredible performance as compared with both existing machines (averaging around IPCs of 1.0 or less) and with most all microarchitectural proposals.

Several elements of the Superspeculative microarchitecture are useful for our own proposed work. First the idea of breaking the data dependence graph of the program is a fundamental new advance for speculative execution and we also want to allow for this to occur within our proposed microarchitectures. The breaking of the control dependence graph of programs turned out to be an enormous performance boost for machines. Likewise, the breaking of the data dependence graph of a program would seem to be at least a significant possible performance enhancing mechanism.

Although the use of current and future value prediction is not likely to yield the very substantial performance gains that have been achieved with conditional branch prediction, we want a microarchitecture that allows for and can accommodate it. We have done this with our Resource Flow execution model approach. Additionally, as the Superspeculative microarchitecture recognized the advantages of predicting instruction dependencies, we likewise recognize the performance problems of waiting for instruction dependencies to be first determined before instruction dispatch. Although we will not predict instruction dependencies in the same way that the Superspeculative microarchitecture does, we will allow for the dispatch of instructions without first determining dependencies. Rather, in our Resource Flow execution approach, we will dynamically determine instruction dependencies during their execution ! Yes, wait and see what we have planned for that ! Like the Superspeculative microarchitecture, our presented microarchitectures are also able to implement any ISA (legacy or new).

2.11.4 The Ultrascalar microarchitecture

The Ultrascalar microarchitecture was first introduced by Henry et al. [47] Their goal was to plan for future machines with very large numbers of component resources. They realized that machines with more component resources while also employing out-of-order execution was a requirement for achieving large ILP extraction in the future. They attempted to come up with a machine microarchitecture that allowed for better management of a large amount of out-order execution. A somewhat notable cliche of theirs summarizes their motivation and goal: "How I learned to stop worrying and love out-of-order execution." This microarchitecture exhibits (according to its authors) a form of machine size scalability known as *asymptotic scalability*. This is a major feature of this microarchitecture that we will discuss more below. This microarchitecture can also be used to implement any ISA (existing or new ones) and therefore does not require any special compiler assistance for its operand or performance enhancing capabilities.

The Ultrascalar machine retains centralized instruction fetch and decode (like most machines, current or proposed) but rather introduces a different way to manage its instruction window. This microarchitecture combines the traditional instruction window (or reservation stations) and execution resources of a more conventional microarchitecture into a new and single set of machine components, which could be termed *execution stations* (ES). This microarchitecture is outlined in Figure 2.12. Figure 2.12 shows the salient elements of this proposed microarchitecture. The execution stations are the components spread out horizontally in the middle of the figure. Instructions are dispatched to these components after they have been fetched, decoded, and have had their data dependencies determined. Each execution station receives a single dispatched instruction and contains the complete logic necessary to execute its instruction. All instructions are dispatched

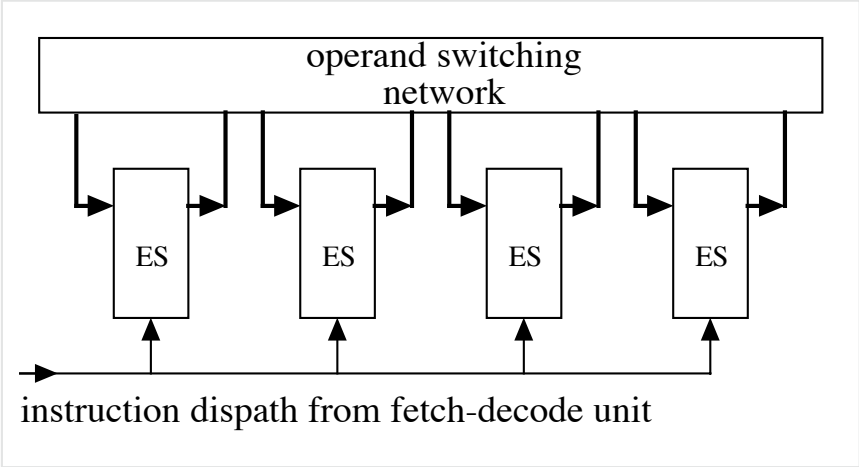

Figure 2.12: *The Ultrascalar Microarchitecture.* The Ultrascalar microarchitecture combines the more traditional instruction window and execution resources into a set of new structures which could be called Execution Stations (labeled ES). An operand switching network serves the purpose of arranging for the output operands from each instruction to be routed as input to the properly dependency instructions.

in-order. All execution stations are identical and any instruction type can be dispatched to any free execution station. Therefore each execution must be able to execute any type of instruction. Instructions within the execution stations are available for execution once their input operands arrive. Instructions remain within an execution station until it is retired (either committed or abandoned). Finally, instructions are dispatched to the execution stations in a control dependent manner. This microarchitecture therefore most closely belongs in the class of distributed machines that employ control dependence decentralization. [104]

The set of all execution stations are arranged in a logical ring. A mechanism is provided for within the microarchitecture to distinguish the equivalent of a head and tail execution station within the ring. This is identical to the arrangement that was previously discussed in the context of the Multiscalar microarchitecture. The execution station at the head of the ring is the one that is next going to be committed, if it gets committed at all (the oldest instruction within the ring). That execution station at the tail of the ring is the one that had an instruction most recently dispatched to it (the youngest instruction in the ring). As instructions get committed the head pointer is advanced to point to the next adjacent execution station. As new instructions are dispatched, the tail pointer is likewise updated to point to the latest dispatched execution station. Empty execution stations are allowed for. Using this structure, the set of all execution stations replaces the need for the more traditional reorder buffer and instead the execution stations themselves serve as the effective reorder buffer. The idea of combining the functions of the instruction window (or reservation stations) and the reorder buffer into a single structure is similar to the Register Update

Unit (RUU) first introduced by Sohi. [132]

This microarchitecture has already introduced a number of new ways to arrange components within a microarchitecture. However, it is how operands are transported from one execution station to the next that makes this microarchitecture more interesting still. The authors introduce the idea of an operand switching network that has an interconnection complexity that grows logarithmically with respect to the number of execution stations employed. With some simplification, the interconnection complexity c grows as $O(\log W)$ where W is the number of execution stations in a design. The network is literally (as might be supposed) composed of a logarithmically hierarchical arrangement of sub-switches that provide for a non-blocking interconnection from each of its inputs to each of its outputs. Referring to Figure 2.12, the interconnection network is simply shown as a block box at the top with bus connections coming from and going to the operand outputs and inputs respectively of each execution station. Note that in general, each execution station (holding a single instruction) generally required two or more inputs (and also possibly provides two or more outputs) so the number of inputs and outputs that the network has to switch is generally greater than one for each execution station. The number of outputs and inputs required is dependent of the particular ISA being implemented. As instructions are dispatched to execution stations, the network is reconfigured to arrange for the proper input operand dependencies to be routed to the newly dispatched instruction from the outputs of preceding instructions that have already been dispatched. The reader is encouraged to refer to the author's work for more details on both the network interconnection arrangement and operation.

One of the difficulties with this microarchitecture is the fact of its still rather large complexity for its interconnection network. Although it grows as order-logarithmic with respect to the number of execution stations, that may still be too large for practical layout considerations in silicon for some time to come. However, it is felt by the authors that a microarchitecture based on the ideas of Ultrascalar will still outperform conventional processors as the numbers of machine hardware components are increased. Another disadvantage of this microarchitecture is that each execution station requires a full execution subunit in order to execute any of the instructions in the given microarchitecture. Although instructions like `add`, *sub*, et cetera, and even those of `load` and `store` can be executed within each execution station, it is neither a good idea nor a necessity that floating point oriented instructions (to name an example of more complicated instructions) should be executed locally within each execution station. The real problem is to do something like a floating point multiply and divide everywhere. Not only do instructions like this take up enormous amounts of silicon area, they represent a substantial waste of silicon in that they are not generally utilized to the degree of their number within Ultrascalar (one per execution station).

In many respects the Ultrascalar microarchitecture most closely resembles that of our own work as compared with many other proposed machines. Design features of our own work that we are

roughly shared with Ultrascalar include:

- control dependent decentralization
- centralized fetch
- in-order instruction dispatch
- merging of the instruction window (or reservations stations) with the reorder buffer into a single component
- the ring structure of the execution stations
- the forwarding of operands from execution station to subsequent execution station without first going through or past a register file

However, in many other ways, our work will differ markedly from that of Ultrascalar. More specifically:

- we use simpler operand interconnection for operand transfers
- we do not embed the hardware for executing all instruction types (like floating point) for each instruction in-flight within the machine
- we do not require the determination of instruction dependencies before instruction dispatch

These are just some of the differences between our work and Ultrascalar.

2.12 Summary

We have presented several microarchitectural developments that have allowed for both parallel and out-of-order execution of instructions from a binary program being executed. We have also presented some other machine microarchitectural proposals and have somewhat related them to our own work where applicable. The various machine features and components that have been briefly presented have allowed for the extraction of instruction level parallelism for even serial sequential workloads that do not feature explicit parallelism at the semantic level of the application, as expressed in the binary program being executed. Our goal is to combine many of the design features of both historical and existing microarchitectures in a new and more generalized way. Some of the microarchitectural execution features that our microarchitectures will implement include: control-flow prediction, value prediction, dynamic microarchitectural instruction predication, dynamic operand dependency determination, and multipath execution. Another goal for our microarchitecture is for

it to facilitate physical scalability of the machine. This means that more microarchitectural components can be included into a machine design while not substantially compromising the difficulty of silicon implementation or the speed of the machine. But the use of more component resources in large machines allows for more ILP extraction from executed programs and correspondingly better execution performance.

In Chapter 3 we present the primary machine component, along with how it functions, that serves as the basis for our new computer (resource flow) execution model. As will be seen, our new basic component builds on the idea of a reservation station, or more specially something more closely resembling an RUU entry. In Chapters 4 and 9 we present entire machine microarchitectures that both draw from elements of the microarchitectures presented in this chapter as well as using the primary component that we propose in Chapter 3.

Chapter 3

The Resource Flow execution model

In this chapter we present the basic concepts employed within the Resource Flow execution model. We also briefly present a key microarchitectural machine component used to implement the core part of this execution model. This machine component is termed the *issue station* (IS). We do not present a complete machine microarchitecture in this chapter, but we do so in subsequent chapters.

The idea of Resource Flow computing is that speculative execution is not constrained by either the control flow graph or the data flow graph of the program, but rather by the available resources within a representative microarchitecture. In effect, only machine structural constraints serve to limit instructions from entering into execution (albeit speculative execution). Additionally, we elevate the importance of instruction operands (whether they be control or data) almost to the level of an instruction itself. Operands are enhanced with additional state that allows for a more unform management of their flow through the machine. This philosophy of execution allows for a large number of simultaneous instruction executions and re-executions as can be structurally sustained on the available machine resources. Executions are allowed to proceed with any available feasible source operands whether predicted or not. The general idea is to first speculatively execute any pending instruction whenever any suitable machine resource is available and then to perform successive re-executions as needed as control and data dependency relationships are determined dynamically during execution.

Our approach attempts to generalize the management and operational issues associated with control-flow prediction, value prediction, and the possible re-execution of those instructions that have already been fetched and dispatched. Our microarchitectural approach also lends itself towards resource scalability through manageable spatial distribution of machine components. A machine that better scales with respect to machine resources, is better able to execute a larger number of instructions simultaneously. Executing more instructions in parallel allows for increases in machine performance not otherwise possible, since expected ILP performance will never be more than the

possible IPC achieved in any machine.

In the remainder of this chapter we first present the basic concepts employed by the Resource Flow execution model. We then discuss the idea of the issue station and briefly outline its basic structure and operation. Next, we briefly address the requirement of interconnecting issue stations among themselves for the purposes of exchanging operands. In that context, we also introduce the idea of *operand filtering units* and their function in isolating physical buses as well as serving to possibly reduce the number of operands that need to be exchanged. This is followed with an example execution of a small code fragment. We then summarize.

3.1 Resource Flow basic concepts

In this execution model the management of operands, including dependency ordering and their transfer among instructions, dominates the microarchitecture. Another primary element of the execution process is the dynamic determination of the control and data dependencies among the instructions and operands. The means and implications of our dynamic dependency scheme is presented. Also, the establishment of a machine component used to hold an instruction during its entire life-cycle after it has been dispatched for possible execution. This component generalizes how instructions are handled during both initial execution and subsequent executions (re-executions) and also eliminates the need for any additional re-ordering components. We term this component an *issue station* (IS). Additionally, some details about how operands are passed from one instruction to another are discussed.

3.1.1 The operand as a first class entity

Within our microarchitectural framework, we identify three types of instruction operands. These are termed *register*, *memory*, and *predicate*.

For predicate operands, two types need to be distinguished. One type is a part of the ISA of the machine and is therefore visible to programmers. A predicate operand of this type would be used, for example, in an ISA such as the iA-64 [54]. For our purposes, this sort of explicit predicate operand is identical to a register operand (discussed above) and is simply treated as such. Not as clear is the use of predicate operands that are not a part of the ISA and are therefore not visible to the programmer. This type of predication is entirely maintained by the microarchitecture itself, but still essentially forms an additional input operand for each instruction. This single bit operand is what is used to predicate the instruction's committed execution, the same as if it was explicit in the ISA. This input predicate thus enables or disables its associated instruction from producing its own new output operand. It should be noted that regardless of the state of

the instruction's predicate (enabled or disabled) an instruction can still be allowed to execute. The real constraint is whether any associated output result operand can be allowed to be passed to subsequent instructions (in program-ordered time). For microarchitectures that support these microarchitecture-only predicate operands, they too can be time-tagged, thus allowing them the same degree of out-of-order flexibility similar to register and memory operands. Finally, note that even ISAs that define predicate registers can also independently employ predication of instructions within the microarchitecture.

3.1.2 Dynamic dependency ordering

Rather then calculating instruction dependencies at instruction dispatch or issue time, we allow instructions to begin the process of executing (possibly with incorrect operands) while also providing for instructions to dynamically determine their own correct operand dependencies. The mechanism used for this dynamic determination of operand dependencies is designed to provide a special tag that conveys the program-ordered relative time of the origin of each operand. This time ordering tag is associated with the operational state of both instructions and operands and is a small integer that uniquely identifies the relative position of an instruction or an operand in program-ordered time. Typically, time-tags take on small positive values with a range approximately equal to the number of instructions that can be held in-flight within an implementation of a machine. The Warp Engine [17] also used time-tags to manage large amounts of speculative execution, but our use of them is much simpler than theirs. Time-tags are present on some recent machines (like the Pentium 4 processor [49]), though they are used for different purposes than as employed in our microarchitecture.

A time-tag value of zero is associated with the in-flight instruction that is next ready to retire (the one that was dispatched the furthest in the past). Later dispatched instructions take on successively higher valued tags. Operands take on the same time-tag values corresponding to the instructions that originate them. Instructions originate operands as outputs from the execution of the instructions. An instruction can also originate an operand by propagating as its own output value an architected operand that existed before it is program-ordered time. This latter situation can occur when an instruction is currently disabled for execution by which still needs to propagate an architected operand for one or more of its outputs. This idea is discussed later within the context of operand forwarding. As instructions retire, the time-tag registers (microarchitectural state holding the time-tag values) associated with all instructions and operands are decremented by the number of instructions being retired. Committed operands can thus be thought of as taking on negative valued time-tags. By comparing time-tag values with one another, the relative program-ordered time relationship is determined. Other variations for the assignment and management of

the time-tag values are also possible.

3.1.3 Handling multipath execution

Multipath execution [148] is a means of speculatively executing down more than one speculative path of a program simultaneously. Multiple speculative program paths originate from control-flow change instructions that have more than a single possible output path for them. The most common of these types of instructions is the common conditional branch. In order to provide proper dependency ordering for multipath execution, we introduce an additional register tag (termed a *path ID*) that will be associated with both instructions and operands on a particular speculative program path.

Generally, a new speculative path may be formed after each conditional branch is encountered within the instruction stream. Execution is speculative for all instructions after the first unresolved conditional branch. At least two options are available for assigning time-tags to the instructions following a conditional branch (on both the taken and not-taken paths).

1. One option is to dynamically follow each output path and assign successively higher time-tag values to succeeding instructions.

2. The second option is to try to determine if the *taken* outcome of the branch joins with the *not-taken* output instruction stream. If a join is determined, the instruction following the *not-taken* output path can be assigned a time value one higher than the branch itself, while the first instruction on the *taken* path would be assigned whatever value it would have gotten counting up from the first instruction on the *not-taken* path.

Note that both options may be employed simultaneously in a microarchitecture. In either case, there may exist instructions in-flight that possess the same value for their time-tag. Likewise, operands resulting from these instructions would also share the same time-tag value. This ambiguity is resolved through the introduction of the path ID. Through the introduction of the program-ordered time-tag and the path ID tag, a fully unique time-ordered execution name space is now possible for all instructions and operands that may be in flight.

3.1.4 Names and renaming

Names for instructions can be uniquely created with the concatenation of the following elements:

- a path ID
- the time-tag assigned to a dispatched instruction

For all operands, unique names consist of:

- type of operand

- a path ID
- time-tag
- address

The type of the operand would be either: *register*, *memory*, or *predicate*. The address of the operand would differ depending on the type of the operand. For register operands, the address would be the name of the architected register. All ISA architected registers are typically provided a unique numerical address. These would include the general purpose registers, any status or other non-general purpose registers, and any possible ISA predicate registers. For memory operands, the address is just the programmer-visible architected memory address of the corresponding memory value. Finally, for our predicate operands, an address value is not used at all since the predicate operand itself has no associated architectural counterpart. Rather, the predicate operand exists only within the microarchitecture itself. However predicate operands still have associated time-tag values. These time-tag values serve to identify the particular predicate register in question. This time-tag value is the same as that associated with the instruction that originates the particular predicate operand. It should be noted that for instruction set architectures that have architectural predicates, they are handled within the Resource Flow execution model identically to how all other architected registers are handled. Due to the novel nature of microarchitectural predicates and their associated complexity beyond that of architectural register and memory operands, a very extensive and detailed discussion of them and their interactions is given in Chapter 7.

Through this naming scheme, all instances of an operand in the machine are now uniquely identified, effectively providing full renaming. All false dependencies are now avoided. There is no need to limit instruction dispatch or to limit speculative instruction execution due to a limit on the number of non-architected registers available for holding temporary results. Since every operand has a unique name defined by a time-tag, all necessary ordering information is thusly provided. This information can be used to eliminate the need for physical register renaming, register update units, or reorder buffers.

3.2 The Issue Station idea

The issue station provides the most significant distinction of this microarchitecture from most others. Still, our issue station and its operational philosophy is very similar to that used by Uht et al. [153], which itself was very similar to their previous work [154]. Our issue stations can be thought of as being reservation stations [141] but with additional state and logic added to them that allows for dynamic operand dependency determination as well as for holding a dispatched instruction (its decoded form) until it is ready to be retired. This differs from conventional reservation stations or issue window slots in that the instruction does not free the station once it is dispatched to a function

unit (FU). Also, unlike reservation stations, but like an issue window slot, the instruction operand from an issue station may be dispatched to different function units (not just one that is strictly associated with the reservation station). The state associated with an issue station can be grouped into two main categories. There is state that is relevant to the instruction itself, and secondly there is state that is related to the operands of that instruction (both source and destination operands). The state associated with the instruction itself has to do with the ordering of this instruction in relation to the other instructions that are currently in the execution window. The remainder of the state consists of one or more input source operands and one or more output destination operands. All operands, regardless of type and whether a source or a destination, occupy a similar structure within an issue station. This structure is termed an *operand block*. The operand blocks all have connectivity to both the operand request and forwarding buses as well as to the FU issue and result buses. More detail on these operand blocks and operand management is provided in the next section.

The state that is primarily associated with the instruction itself consists of the following:

- instruction address
- instruction operation
- execution state
- path ID
- time ordering tag
- predication information

The *instruction operation* is derived from the decoded instruction and specifies the instruction class and other details needed for the execution of the instruction. This information may consist of subfields and is generally ISA specific. The *instruction address* and *predicate information* are only used when dynamic predication [95] is done within the microarchitecture. The *path ID* value is used when dynamic multipath execution is done. The *time-tag* value is used to order this instruction with respect to all others that are currently within the execution window of the machine. The *execution state* value consists of a set of bits that guide the execution of the instruction through various phases. Some of this state includes:

- in the process of acquiring source operands for first time
- execution is needed
- in the process of executing (waiting for result from FU)
- at least one execution occurred
- a result operand was requested by another issue station
- a result operand is being forwarded

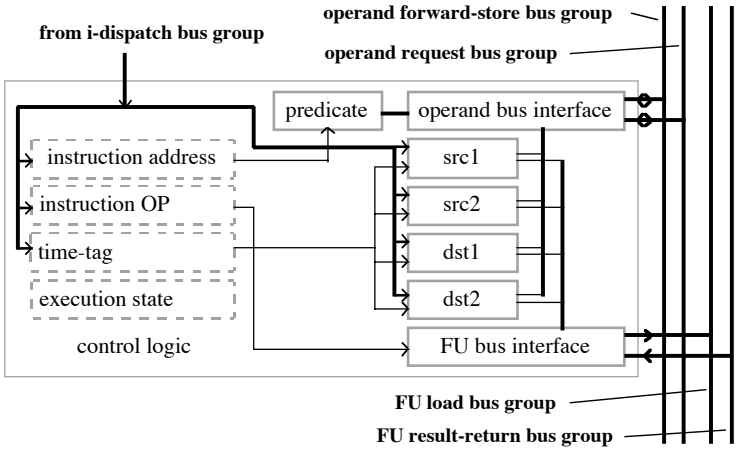

Figure 3.1: *High-level block diagram of our issue station.* The major state associated with an issue station is shown: four operand blocks (two source and two destination) and its four bus interfaces, grouped according to bus function into two major logic blocks.

In addition to guiding the operation of the issue station, many of these state bits are used in the commitment determination for this issue station.

A simplified block diagram of our issue station is shown in Figure 3.1. The state associated primarily with just the instruction is shown on the left of the block diagram, while the operand blocks and their connectivity to the various buses is shown on the right. In this example, a total of four operand blocks are shown, labeled: *src1*, *src2*, *dst1*, and *dst2*. The number of source and destination operand blocks that are used for any given machine is ISA dependent. Some ISAs require up to five source operands and up to three destination operands (usually for handling double precision floating point instructions). Generally, any operand in the ISA that can cause a separate and independent data dependency (a unique dependency name) requires a separate operand block for it. No additional operand block is needed when dynamic predication is performed since its state would be included as part of the general issue station state mentioned previously.

3.2.1 Register and memory operand storage

Register and memory operands are similar enough that they share an identical operand block within the issue station. The state within an operand block consists of:

- type of operand
- path ID
- time ordering tag
- address
- size

- previous value
- value

The operand *type*, *path ID*, and *time ordering tag* serve an analogous purpose as those fields do within an issue station, except that these fields now apply specifically to this particular operand rather than to the instruction as a whole.

The *address* field differs depending on the type of the operand. For register operands, the address would be the name of the architected register. These would include the general purpose registers, any status or other non-general purpose registers, and any possible ISA (architected) predicate registers (like those in the iA-64 ISA [54, 119]. For memory operands, the identifying address is just the programmer-visible architected memory address of the corresponding memory value.

Although the time ordering tag uniquely identifies the issue station that forwarded the operand, it does not indicate information about a particular instance of that operand being forwarded. The *sequence number* is used to disambiguate different instances of an operand forwarded with the same time-tag. This is only needed when more elaborate forwarding interconnection fabrics are used that allow an operand to either get duplicated in-flight or to pass and overtake another operand in real time. The sequence number is not used in the present work.

The *size* is only used for memory operands and holds the size of the memory reference in bytes. The *value* holds the present value of the operand, while the *previous value* is only used for destination operands and holds the value that the operand had before it may have been changed by the present instruction. The previous value is used in two possible circumstances. First, it is used when dynamic predication is employed and the effects of the present instruction need to be squashed (if and when its enabling predicate becomes false.) It is also used when a forwarded operand with a specific address was incorrect and there is no expectation that a later instance of that operand with the same address will be forwarded. This situation occurs when addresses for memory operands are calculated but are later determined to be incorrect. An operand with the old address is forwarded with the previous value to correct the situation. Figure 3.2 shows a simplified block diagram of an operand block.

3.2.2 Operand forwarding and snooping

As with microarchitectures that do not use time-tags, operands resulting from the execution of instructions are broadcast forward for use by waiting instructions. As expected, when operands are forwarded, not only is the identifying address of the operand and its value sent, but also the time-tag and path ID (for those microarchitectures using multipath execution). This tag will be used by subsequent instructions (located later in program-order time) already dispatched to determine

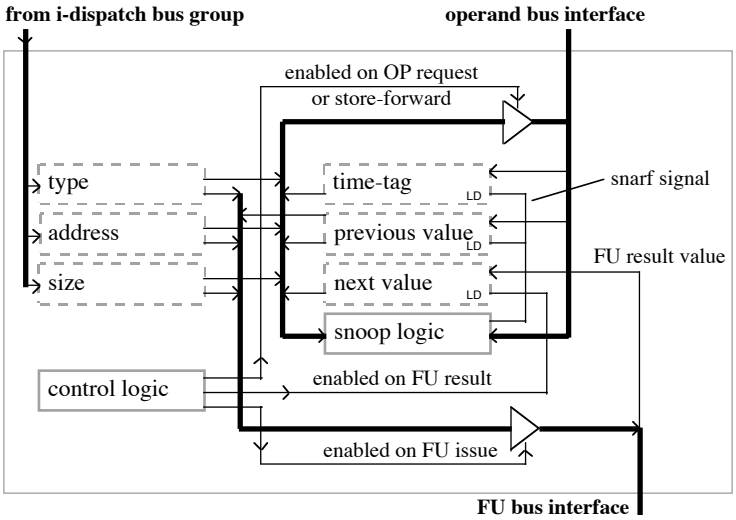

Figure 3.2: *Block diagram of an Operand Block.* Each Operand Block holds an effectively renamed operand within the issue stations. Several operand blocks are employed within each issue station depending on the needs of the ISA being implemented. The state information maintained for each operand is shown.

if the operand should be *snarfed* [1] as an input that will trigger its execution or re-execution.

The information associated with each operand that is conveyed from one instruction to subsequent ones is referred to as an *operand transfer*. The name connotes the fact that an operand along with its associated metadata is transferred from one issue station to others on some sort of interconnection fabric (often a set of parallel buses). We have also sometimes termed this transfer a *transaction* (from the idea of a bus transaction). The information within an operand transfer generally consists of:

- transfer type
- operand type
- path ID
- time-tag of the originating instruction
- identifying address
- data value for this operand

This above information is typical of all operand transfers. True flow dependencies are enforced through the continuous snooping of these transfers by each dispatched instruction residing in an issue slot that receives the transfer. Each instruction will snoop all operands that are broadcast to it but an operand forwarding interconnect fabric may be devised so that operand transfers are only sent to those instructions that primarily lie in future program-ordered time from the originating

[1] snarfing entails snooping address/data buses, and when the desired address value is detected, the associated data value is read

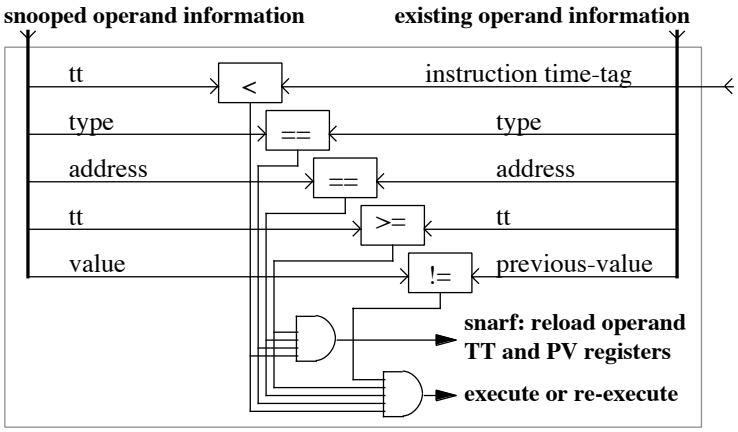

Figure 3.3: *Source Operand Snooping.* The registers and snooping operation of one of several possible source operands is shown. Just one operand forwarding bus is shown being snooped but typically several operand forwarding buses are snooped simultaneously.

instruction. More information on operand forwarding interconnection networks is presented in a later section. More details on operand transfers for register, memory, and predicate forward operations are also presented later.

Figure 3.3 shows the registers inside an issue slot used for the snooping and snarfing of one of its input operands. The *time-tag*, *address*, and *value* registers are reloaded with new values on each snarf, while the *instruction time-tag* is only loaded when an instruction is dispatched. The operand shown is typical for a source register, a source memory operand, or an instruction execution predicate register.

If the path ID and the identifying address of the operand matches any of its current input operands, a check is also made to see if the time-tag value is less than its own assigned time-tag and greater than or equal to the time-tag value of the last operand that it snarfed, if any. If the snooped data value is different than the input operand data value that the instruction already has, a re-execution of the instruction is initiated. This simple rule will allow for the dynamic discovery of all program dependencies during instruction execution while also allowing for maximum concurrency to occur.

3.3 Result forwarding buses and operand filtering

In this section we discuss how operands are exchanged among issue stations and other units that may reside in the flow path of operands during their lifetimes. Examples of machine components other than issue stations that may need to receive operands are the architected register file (if the microarchitecture has one) or the load-store queue (LSQ). In order to provide the necessary input

operands to satisfy dependencies for subsequent instructions, operands must be forwarded to issue stations that lie in future program-ordered time. Additionally, requests for input operands must travel backwards in program-ordered time to those issue stations executing previous (in program order) instructions. In the following subsections, we briefly discuss options for use in creating the operand interconnection fabric, the idea of employing electrical repeater units to physical break long physical buses into shorter ones. Finally, we briefly discuss some details of the actual information that is transferred on the interconnection fabric.

3.3.1 Interconnection bus fabrics

There are several choices for a suitable interconnection fabric. A simple interconnection fabric could consist of one or more shared buses that simply interconnect all issue slots. For machines employing more than a small number of issue stations (for example, eight or more) a group of buses operating in parallel may be required since the bandwidth of a single bus will likely be exceeded. Although appropriate for smaller sized microarchitectures, this arrangement does not scale as well as some other alternatives. Specifically, a single span of a physical bus need not interconnect all of the issue stations within a given microarchitecture. A more appropriate interconnection fabric that would allow for the physical scaling of the microarchitecture may be one in which buses are segmented with active repeaters between physical bus stages. This arrangement exploits the fact that register lifetimes are fairly short [33, 133]. Registers being forwarded to instructions lying close in future program-ordered time will get their operands quickly while those instructions lying beyond the repeater units will incur additional delays.

Another possibility for the operand forwarding fabric is to have separate buses for each type of operand. This sort of arrangement could be used to tailor the available bandwidth provided for each type of operand. It would also allow for a different interconnection network to be used for each type of operand. Further, separate buses may be used for each direction of operand transfer (for the forwarding and backwarding of operands, respectively). Of course, besides using buses, any number of alternative switching arrangements are possible. Some of the ideas presented here for interconnections are used in our new microarchitectures presented later (Chapters 4 and 9).

In addition to allowing for physical scaling, the idea of employing repeater units to separate physical buses also offers the opportunity for filtering out some operands that do not need to be forwarded beyond a certain point. This is discussed in the next section.

3.3.2 Operand filtering

The opportunity to provide active repeaters between forwarding bus segments also opens up a range of new microarchitectural ideas not easily accomplished without the use of time-tagging. Different

types of active repeaters can be optionally used. Further, different types of repeaters can be used for the various types of operands. Some can just provide a store-and-forward function while another variation could also store recently forwarded operands and filter out (not forward) newly snooped operands that have already been previously forwarded. The latter type of forwarding repeater unit is termed a *filter unit*. This feature can be used to reduce the operand traffic on the forwarding fabric and thus reduce implementation costs. For example, for memory operands, a cache (call it a *L0 cache*) can be situated inside a *memory filter unit* (MFU) and can hold recently requested memory operands. This can reduce the demand on the L1 data cache and the rest of the memory hierarchy by satisfying some percent of memory operand loads from these caches alone. Register filter units (RFUs) are also possible and can reduce register transfer traffic similarly to the MFUs.

3.3.3 Operand forwarding strategies and bus transfers

Although we have so far described the operand forwarding mechanism in simple terms as being the broadcasting of operands to those instructions with higher valued time-tags, there are some additional details that need to be addressed for a correctly working forwarding solution. These details also differ depending on type of operand. There are many possible strategies for the forwarding of operands (and of operands of varying types). We now briefly outline three such strategies. One of these is suitable for registers. Another is suitable for registers or memory operands. The third is oriented for the forwarding of predicates. These three strategies are termed *relay forwarding*, *nullify forwarding*, and *predicate forwarding*, respectively. In general, each forwarding strategy employs bus transfers of one or more types to implement its complete forwarding solution.

Relay forwarding

This forwarding strategy is quite simple but is also sufficient for the forwarding of register operands within the execution window. The name of this forwarding strategy is derived from the fact that operands and updates to those operands possibly get relayed through intermediate instructions that had also forwarded a value for the given operand. In this strategy, when a new register operand needs to be forwarded from an instruction, the standard operand information is packaged up into what is termed a *register-store* type of operand transfer. This operand transfer type consists of:

- transfer type of *register-store*
- path ID
- time-tag of the originating instruction
- register address
- value of the register

A request is made to arbitrate for an outgoing forwarding bus and this information is placed on the bus when it becomes available.

When an instruction obtains a new input operand, it will re-execute producing a new output operand. In this forwarding strategy, the new output operand is both stored locally and is sent out on the outgoing forwarding buses to subsequent (higher time-tag valued) instructions. Previous values of the instruction's output operand are also snooped as if they were input operands and are also stored locally. It should also be noted that if the enabling execution predicate for the current instruction changes (either from being enabled to disabled or visa versa), a new output operand is forwarded. If the instruction enabling predicate changes from disabled to enabled, the output operand that was computed by the instruction is forwarded. If the instruction predicate changes from enabled to disabled, the previous value of the output operand (before being computed by the current instruction) is forwarded. That previous value is available to the instruction because it gets snooped as if it was an additional input to the present instruction. All output operands of an instruction are snooped for new input values so that they are available to be forwarded in the case when the present instruction becomes disabled. While the present instruction is disabled, all newly snarfed input operands corresponding to output operands of the present instruction are forwarded. By forwarding the previous value of an output operand when an instruction becomes or is disabled, the present instruction acts as an operand relay for some previous instruction. This gives rise to the name of this forwarding strategy.

With this strategy, instructions that are located in the program-ordered future will eventually always get the correct value. The correct value is that which ends up being the committed value if the current instruction also ends up being committed itself (being predicated to execute at commitment determination). This is an elegant forwarding strategy and the simplest of the forwarding strategies investigated so far, and is a reasonable choice for the handling of register operands. The inclusion of the time-tag in the operand transfer is the key element that allows for the correct ordering of dependencies in the committed program.

Nullify forwarding

There are limitations to the applicability of the previously discussed forwarding strategy (relay forwarding). That strategy depends upon the fact that the address of the architected operand does not change during the lifetime of the instruction while it is executing. For example, the architected addresses for register operands do not change for instructions. If the instruction takes as an input operand a register *r6*, for example, the address of the operand never changes for this particular instruction (it remains *6*). This property is not generally true of memory operands. The difficulty with memory operands is that many memory related instructions determine the address

of a memory operand value from an input register operand of the same instruction. Since we allow for instructions to execute and re-execute on entirely speculative operand values, the values of input register operands can be of essentially any value (including a wildly incorrect value) and thus the address of a memory operand can also change while the instruction is in-flight. This presents a problem for the correct enforcement of memory operand values and the dependencies among them. In the case of a memory store instruction, when it re-executes and acquires a new memory store value, the address of that memory store may also have changed. We cannot simply forward that new memory operand (address and value) as with the relay forwarding strategy above. The reason the new memory operand cannot be simply forwarded is because it may have a different architected address and the simple forwarding of it would not supersede the previous memory operand (with its different architected address). Rather, we need some way to cancel the effect of any previously forwarded memory operands. This forwarding strategy does just that.

In this strategy, memory operands that must be forwarded employ a similar operand transfer as above for registers (as described in the context of relay forwarding), but instead have an operand transfer type of *memory-store*. This new operand transfer type includes the memory operand address as well as its value (along with the path and time-tag information). However, when an instruction either re-executes or its enabling predicate changes to being disabled, a different type of forwarding operand transfer is sent out. This new type of operand transfer is termed a *memory-nullify* and has the property of nullifying the effect of a previous store types of operand transfers to the same architected operand address. Although this type of operand forwarding could be applied in the case of register operands, it is not needed in that case and neither has been done for that. This type of operand transfer consists of:

- operand transfer type of *memory-nullify*
- path ID
- time-tag of the originating instruction
- memory operand address
- value of the memory operand

When this operand transfer is snooped by subsequent instructions, for those instructions that have a memory operand as an input (those instructions that load memory values in one way or another), a search is made for a match of an existing memory operand. If a match is detected, the time-tag of that particular memory operand is set to a state such that any future *memory-store* operand transfer, regardless of its time-tag value, is accepted (snarfed). Further, on reception of this *memory-nullify* operand transfer a request is sent backwards in program order for a memory operand with the desired memory address. The operand transfer that represents a request for a memory operand would consist of:

- operand transfer type of *memory-request*

- path ID
- time-tag of the originating instruction
- memory operand address

Of course, the memory address for the operand desired must be in the operand transfer but it is not as obvious why the originating instruction's time-tag is also included. In some interconnection fabrics, the time-tag is included in backwarding requests to limit the scope of the travel of the operand transfer through the execution window. This same scope-limiting function is usually performed for forward going transfers as well. When the request is sent backwards in program order, previous instructions or the memory system itself will eventually snoop the request and respond with another *memory-store* operand transfer. As discussed, this forwarding strategy is very useful for memory operands but it can also be used for register operands with appropriate changes to the applicable operand transfer elements. Again, the inclusion of a time-tag value is what allows for proper operand dependency enforcement in the committed program.

Predicate forwarding

There are several ways in which instructions can be predicated in the microarchitecture. Some of these techniques are discussed in the papers by Uht et al [155] and Morano [96]. Predicate register values are essentially operands that need to be computed, evaluated, and forwarded much like register or memory operands. Each instruction computes its own enabling predicate by snooping for and snarfing predicate operands that are forwarded to it from previous instructions from the program-ordered past. Depending on the particular predication mechanism used, relay forwarding (described above) may be a suitable choice (if not a good one) for handling the forwarding of predicate operands. However, some predication mechanisms need additional operand transfer types (besides a base *store* operand transfer type) in order to function properly. For example, the predication strategy described by Morano [96] requires three operand transfer types in order to be fully implemented and is used as the example in this section for discussing the forwarding of predicate information.

This predication strategy requires two store-type operand transfers rather than just one (as with register or memory operands). These two operand transfers are similar to the previous operand store transfers, but one of these holds two values rather than just one. The first of these is the *fall-through-predicate-store* operand transfer and consists of:

- operand transfer type of *fall-through-predicate-store*
- path ID
- time-tag of the originating instruction
- fall-through predicate value

This operand transfer is analogous to a register or memory store type transfer, but is used instead to forward a single bit value (the current *fall-through predicate* for instructions following the instruction that forwarded the operand). A fall-through predicate is a single bit that specifies the execution status (enabled or disabled) for instructions that lie beyond the not-taken output path (or *fall-through* output path) of a conditional branch. This particular operand transfer could be forwarded by either a conditional branch or by a non-branch instruction. In the case of a non-branch instruction, the only predicate value that makes sense to forward is the its own enabling predicate, and so only one value needs to be forwarded.

In the case of a conditional branch instruction, there are two possible output predicates that must be considered: 1) for the taken output path and 2) for the not-taken path. In order, to forward both values for these instructions, to program-ordered future, the second store operand transfer type (mentioned previously) is used. This operand transfer, termed a *branch-target-predicate-store*, consists of:

- operand transfer type of *branch-target-predicate-store*
- path ID
- time-tag of the originating instruction
- branch target instruction address
- fall-through predicate value
- branch target predicate value

This is identical to the previous *fall-through-predicate-store* operand transfer but also includes the instruction address for the target of the conditional branch (the *taken* address) and the single bit predicate governing the execution status for instructions following the target of the conditional branch in program-ordered future.

Finally, a third operand transfer is used to invalidate a previously forwarded branch target predicate. This operand transfer is a *branch-target-invalidation* and consists of:

- operand transfer type of *branch-target-invalidation*
- path ID
- time-tag of the originating instruction
- branch target instruction address
- time-tag of target predicate to be invalidated

This is similar to other such invalidation operand transfers in that when it is snooped by instructions in the program-ordered future, a search is made for some state (in this case some predicate state) that matches the given criteria within the operand transfer itself. The inclusion of the second time-tag in this operand transfer allows for certain efficiencies that are particular to the predication mechanism described.

TT	instruction	t1	t2	t3	t4
0	r3 := 1	r3=1			
1	r4 := r3 + 1		r3=1 r4=2		
2	r3 := 2			r3=2	
3	r5 := r3 + 2		r3=1 r5=3		r3=2 r5=4

Figure 3.4: *Example Instruction Execution.* The time-tags for sequential program instructions are on the left. Real time is shown advancing along the top. For each real time interval, input operands are shown above any output operands.

3.4 Example execution

In many ways, all operands (whether they be registers, memory, or execution predicates) require the use of time-tags to determine the relative ordering of events in a microarchitecture that otherwise lets all instructions execute and re-execute wildly out of order, in real time, with respect to each other. A simple execution example using time-tags is shown in Figure 3.4. In this example we show how register operands are created and snarfed in real time. Four instructions are listed along with the time-tag (TT) assigned to them on the left. Real time progresses to the right and four time periods are identified. In time period *t1*, the instruction with TT=0 executes and creates its output operand *r3*. This operand is forwarded to succeeding instructions in program ordered future time. In time period *t2*, instructions at TT=1 and TT=3 have snarfed this operand since it was one of their inputs and met the snarfing criteria. These two instructions execute in parallel and create their output operands. Of course, the output for instruction at TT=3 is incorrect but that can not be determined at this point in real time. In time period *t3*, instruction at TT=2 executes creating its output operand. That operand gets forwarded and is snarfed by the instruction at TT=3 because it met the snarfing criteria. That instruction re-executes as a result in time period *t4*, thus creating its correct output. All instructions are now ready for commitment with their correct outputs.

3.5 Summary of Resource Flow execution

We have briefly introduced the Resource Flow execution model. This model allows for the speculative execution of instructions in parallel without the need (and associated delays) for any instruction control or data flow dependencies to be either determined or satisfied. Generally, instructions are

allowed to enter into execution as long as machine execution resources are available (execution is structurally constrained). Speculative operands are used for speculative instruction executions and re-executions are employed as required to finally converge to the proper architected machine state. Convergence is guaranteed by the fact that the last input operands of any instruction come from a previously committed instruction, and are therefore the correct input operands for the present instruction. We have also introduced the key machine component, the issue station, that facilitates the realization of the Resource Flow execution ideas. We presented the internal structure of this component as well as some of its required operations. The development of the idea of the operand as a first-class element of the machine has also been presented and some of the elements that make up operands (of varying types: register, memory, and predicate) have been described. We have also developed and introduced the idea of using time-tags for program dependency ordering.

We also introduced some ideas for how issue stations might be interconnected. In this context we introduced the idea of physically splitting long buses into shorter segments (called spans). This necessitates a bus repeater unit of some kind and this unit might be further enhanced to filter out unnecessary operands that do not require forwarding. These filter units may also serve as a small cache for the quick retrieval of operands. This type of cache function likely speeds up overall machine performance just as convention caches in the memory hierarchy do. We also briefly outlined some details about what information is transferred with operands and some strategies for how to manage proper operand forwarding and acquisition by issue stations.

As might be expected from the discussion in this chapter, the Resource Flow execution model deals with the management and flow of instructions and operands within a machine in a way that allows for the maximum utilization of the available machine resources. This idea of flow computing will become even more evident when we introduce microarchitectures using this execution model.

Chapter 4

OpTiFlow: A Resource Flow microarchitecture

In this chapter we develop and present an introduction to a representative microarchitecture that implements the Resource Flow execution model, that was presented in Chapter 3. This microarchitecture is a relatively simple example that is oriented towards a machine with approximately the same physical silicon size (numbers of transistors) and numbers of components as a current or somewhat next-generation state-of-art processor. This is by no means the only microarchitecture that might implement the Resource Flow execution model ideas, but it is one that is sufficient to show both a working example and what might be expected to be a base for future work and future microarchitectures. The name of this microarchitecture (*OpTiFlow*) is derived from the idea that it employs the flow of operands through program-ordered time (Operand Time Flow). Although this name might be used to describe any microarchitecture that implements the Resource Flow execution model, it may be more evident or understandable in this simple microarchitecture rather than more complications ones (like the one we will introduce in a subsequent chapter).

Consistent with one of our stated goals at the outset, this microarchitecture allows for the implementation of any existing (legacy) instruction set architecture (ISA). This microarchitecture is similar to conventional microarchitectures in many respects. We will discuss the more familiar aspects of the microarchitecture first and then present the more distinctive part of the microarchitecture in subsequent sections.

4.1 General microarchitecture overview

In this section we discuss the parts of the microarchitecture that are identical or similar to many conventional microarchitectures. These parts are basically the memory hierarchy, the instruction

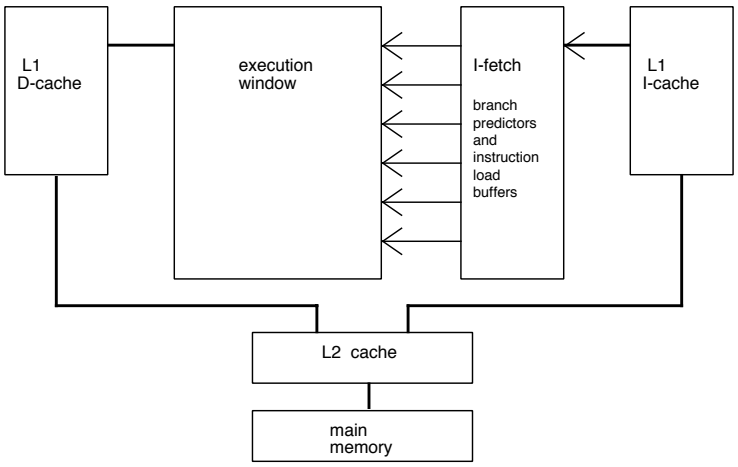

Figure 4.1: *High-level View of a Resource Flow Microarchitecture.* Shown are the major hardware components of the microarchitecture. With the exception of the execution window block, this is similar to most conventional microarchitectures.

fetch mechanisms, branch prediction, instruction decode and dispatch.

The L2 cache (unified in the present case), and the L1 instruction cache are both rather similar to those in common use. Except for the fact that the main memory, L2 cache, and L1 data cache are generally all address-interleaved, there is nothing further unique about these components. The interleaving is simply used as a bandwidth enhancing technique and is not functionally necessary for the designs to work.

The I-fetch unit first fetches instructions from i-cache along one or more predicted program paths. Due to our relatively large instruction fetch bandwidth requirement the fetching of several i-cache lines in a single clock is generally required. Instructions are immediately decoded after being fetched and any further handling of the instructions is done in their decoded form. Decoded instructions are then staged into an *instruction dispatch buffer* so that they are available to be dispatched into the *execution window* when needed. The execution window is where these microarchitectures differ substantially from existing machines. This term describes that part of the microarchitecture were the issue stations and processing units are grouped. The instruction dispatch buffer is organized so that a large number of instructions can be broadside loaded (dispatched) into the issue stations within the execution window in a single clock.

Figure 4.1 provides a high-level view of this microarchitecture. Instructions can be dispatched to issue stations with or without initial input source operands. Once a new instruction is dispatched to an issue station (IS), the issue station begins the process of contending for execution resources that are available to it (depending on microarchitectural implementation). Operand dependency

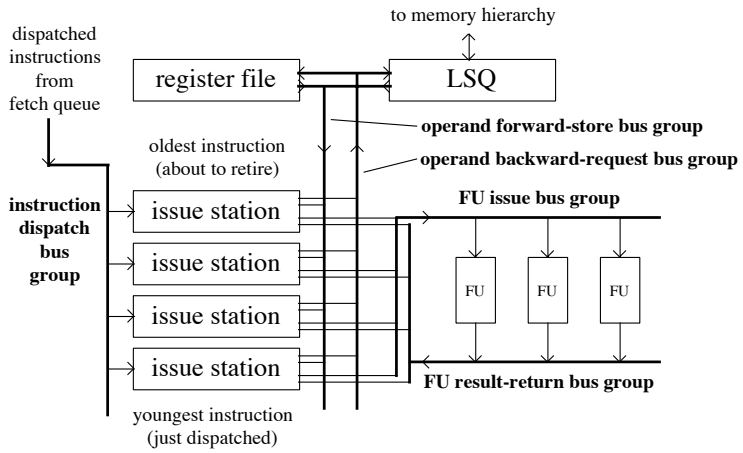

Figure 4.2: *High-level block diagram of the OpTiFlow microarchitecture.* Issue stations are shown on the left and various function units on the right. An architected register file and a load-store-queue is shown at the top. Bidirectional operand request and forwarding buses are shown vertically oriented (to the right of the issue stations). Buses to transport an instruction operation and its source operands to the function units are also shown. Likewise buses to return result operands are present.

determination is not done before either instruction dispatch (to an issue station) or before instruction operation issue to an execution unit. All operand dependencies are determined dynamically through snooping after instruction dispatch. As stated previously, instructions remain in their IS executing and possibly re-executing until they are ready to be retired. All retirement occurs in-order.

4.2 Execution core of OpTiFlow

In addition to a fairly conventional memory hierarchy and fetch unit, this microarchitecture also has a load-store-queue (LSQ) component, an architected register file, execution function units (FU). and the issue station components discussed previously. Decoded instructions are dispatched from the fetch unit to one or more issue stations when one or more of them are empty (available for dispatch). Instructions are dispatched in-order. The number of instructions dispatched in any given clock cycle is the lesser of the number of ISes available and the dispatch width (as counted in numbers of instructions).

Figure 4.2 shows a block diagram of the execution window for this microarchitecture. In the top left of the figure is the architected register file. Since all operand renaming is done within the issue stations, only architected registers are stored here. In the top right is the load-store-queue. The lower right shows (laid out horizontally) a set of function units. Each function unit (three are shown in this example) is responsible for executing a class of instructions and can have

independent pipeline depths. The function unit pipelining allows for new operations to arrive on each successive clock cycle while generating new results on each cycle. Instructions not executed by a function unit are executed within the issue station itself. This currently includes all control-flow instructions, as well as load-store instructions. The lower left shows (laid out vertically) the set of issue stations (four are shown in this example) that may be configured into an implementation of the machine. The ISes are all identical without regard to instruction type. This allows for all instructions to be dispatched in-order to the next available stations without further resource restrictions or management.

Shown running vertically in the center of Figure 4.2 are two buses. These bidirectional and multi-master buses form the means to request and forward operands among the ISes, register file, and LSQ. Each of these buses is actually a parallel set of identical buses that are statistically multiplexed to increase operand transfer bandwidth. Statistical multiplexing of the buses means that any free bus-transfer clock period is available for handling any available operand that is waiting for transport. Operands waiting for bus-transfer clock slots are not assigned (like for example by an attribute of the operand itself) to any specific bus in the group, but rather can use any of the buses that may have an available transfer slot. Other bus arrangements are possible (some fairly complicated by comparison). Further, separate bus fabrics for handling different types of operands is also possible. One of these buses is used by ISes for requesting source operands and has been termed a *backwarding request* bus. The name is derived from the fact that requested operands should only be satisfied by those instructions that lie in the program-ordered past from the instruction requesting the operand. The other bus is used for forwarding operands to younger instructions and is often termed the *forwarding* bus. Operands need to be forwarded from older dispatched instructions to younger dispatched instructions. The arrows on the buses show the direction of intended travel for operands (or operand requests) that are placed on each bus respectively. Although the register file and LSQ only receive operand requests on the backwarding request bus, the ISes both receive and transmit requests from their connections to that bus. Likewise, although the register file and LSQ only transmit operands on the operand forwarding bus, the ISes both receive and transmit on their connections to that bus. The number of ISes in any given machine implementation roughly corresponds to the number of elements of a reorder buffer (ROB) or a register update unit (RUU) in a more conventional machine. It should also be noted that both the LSQ and the architected register file of this design employ operand snooping logic very similar to that of issue stations, as presented in Chapter 3. In fact, all of the components that pass operands amongst themselves, and which are connected to the operand interconnection fabric, employ the same basic logic to snoop for input operands, snarf input operands, snoop for operand requests from other components, and respond to requests from other components.

This microarchitecture also employs the use of function units to perform the actual execution of

decoded instructions. These are shown in the bottom right area of Figure 4.2. Function units are relatively small execution units (generally pipelined) that only perform a subset of the ISA of the given machine architecture. The use of function units generally allow for more efficient utilization of them over a general purpose unified execution unit (each of which can execute any instruction in the given ISA). And that is the reason that they were employed there. However, this particular microarchitecture does not preclude the use of unified execution units. The function units in this microarchitecture are not too dissimilar to those of conventional microarchitectures. They are rather conventional execution pipelines with the exception that a tag representing the originating issue station flows through the pipeline along with the intermediate execution results.

In this microarchitecture, unidirectional buses are provided to interconnect the ISes to the FUs. The buses to and from the FUs are generally multiple identical buses in parallel to allow for increased transfer bandwidth. It is assumed that all buses carry out transfers at the same clock rate as the rest of the machine including the execution function units. One bus group serves to bring decoded instructions along with their input source operands from an issue station to a function unit. The other bus returns function unit results (one or more result operands) back to its originating issue station. Outgoing (IS to FU) requests are tagged with the time-tag of the originating IS. In this way, the results are properly reacquired by the originating IS after the completion of the execution within the FU. Since the FUs are pipelined, output results from an FU may not arrive back to its originating IS for several clocks. The process of acquiring an FU execution slot through arbitration also can cause additional clock period stalls for IS execution requests. Finally, depending on bus and logic propagation delays, additional clock periods are used to get an IS request to an FU and back to its originating IS again. The present microarchitecture incurs a one clock delay for these bus transfers. The number and types of function units can vary in the same manner as in conventional machines.

This particular microarchitecture is oriented towards implementing the Resource Flow ideas while having an overall machine size similar to existing high-end processors or to that of next generation processors. Specifically, large scalability of the machine was not a goal with this design. However, a future microarchitecture that we present in a subsequent chapter will indeed address the scalability issue.

4.3 Summary

We have described a new microarchitecture that combines some of the features of conventional superscalar microarchitectures along with those of a value-predicting microarchitecture. This microarchitecture is one which also implements the ideas embodied in the Resource Flow execution

model, as presented in Chapter 3. This microarchitecture enables a new kind of *flexible* instruction execution parallelism to an extent and degree not possible in other microarchitectures. We have employed the use of issue stations along with an architected register file, a load-store-queue, and function units to realize a machine with the approximate size and component constraints (in terms of numbers of components) of current or near term machines. Although a simple operand interconnection fabric has been employed (sets of parallel buses), it provides the means to allow for the necessary exchange of operands among the issue stations, the architected register file, and the load-store-queue. This has all been achieved while still maintaining binary program compatibility with existing ISAs.

A subsequent chapter provides a basic characterization of this microarchitecture. However, before that could be done some sort of simulation framework was needed to capture and simulate in a rather detailed way the basic operation of this microarchitecture (through the simulation of its various hardware components and their interconnections). This simulation framework was developed and is presented in the next chapter.

Chapter 5

Simulation methodology for OpTiFlow

Due to the very distributed nature of the OpTiFlow microarchitecture, and indeed to all Resource Flow microarchitectures, some care needs to be given to the problem of modeling the microarchitecture to the extent that most or all of the dynamics of the machine can be implemented. The difficulty is that many existing simulation frameworks were not designed for arbitrarily complex distributed state machines and are therefore quite unsuitable for use as a basis for the microarchitectural modeling of OpTiFlow. In this chapter we present a simulation framework that is sufficiently flexible to model the distributed nature of the OpTiFlow microarchitecture.

We first present some of the difficulties associated with the modeling of the OpTiFlow microarchitecture as compared with more conventional microarchitectures. We also review some of the more common microarchitectural simulation frameworks and explain why they are inadequate for our purposes. We then present an outline of our simulation framework and how it has the flexibility to model what we need. Next we discuss some of the implementation choices available for our simulator. These include (among other things) the choice of the instruction set architecture (ISA) to simulate, the choice of the operating system (OS) to compile programs against, and the way in which we trap system calls in order for them to be implemented. We finally summarize in the last section.

5.1 Modeling OpTiFlow

In this section we present some of the difficulties with simulating the OpTiFlow microarchitecture as compared with many more conventional microarchitectures. All computers are basically large finite state machines (FSM). Systematically they perform on every clock period a combinatorial logic computation that takes as input the existing state of the machine (and perhaps some new inputs) and generates a new state. This new state is calculated with the necessary setup time to

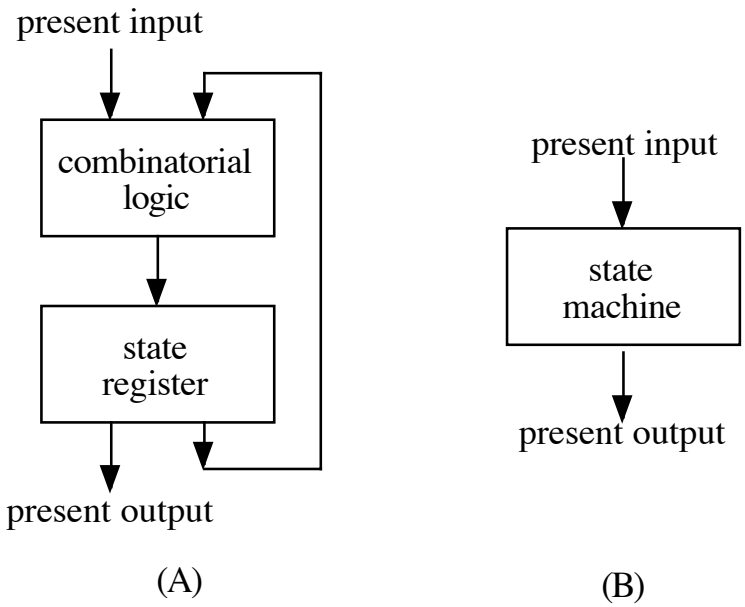

Figure 5.1: *The basic structure of a Moore-type finite state machine.* Part A show a single unified combinatorial logic block that takes the present state (as it exists on the output state register) and the present input and calculates the next state. After the next clock transition, the next state becomes the new present state. Part B shows the same machine in abbreviated form.

the next clock transition, and upon that next transition the calculated next state becomes the new present state. Of course, this process occurs on a huge scale and without some underlying modular structure it would be otherwise entirely incomprehensible. For reference, part `(A)` of Figure 5.1 shows a finite state machine (of the Moore variety) that also can be used to basically depict the operation of a computer. Part `(B)` of the same figure shows an alternate (abbreviated) view of the same state machine as depicted in part `(B)`.

If the only conceptual view of the operation of a computer was that of Figure 5.1 computer designers would indeed have many problems on their hands other than trying to figure out new microarchitectural machine organizations. However, most conventional microarchitectures actually exhibit a very structured state machine model that more closely resembles that of Figure 5.2. This latter figure shows two finite state machines in a pipeline arrangement where the outputs of one feed into the inputs of the second. Each of these finite state machines (forming a pipeline stage) is essentially that depicted in Figure 5.1. Outputs from the final stage loop around to become some of the inputs of the first stage. This figure (Figure 5.2) more closely resembles the pipelined nature of many modern computer microarchitecture. Although most modern processor designs feature many pipeline stages [50], for simplicity Figure 5.2 only shows two pipeline stages. Remarkably, the model of Figure 5.2 is close enough to the reality of many conventional microarchitectures that

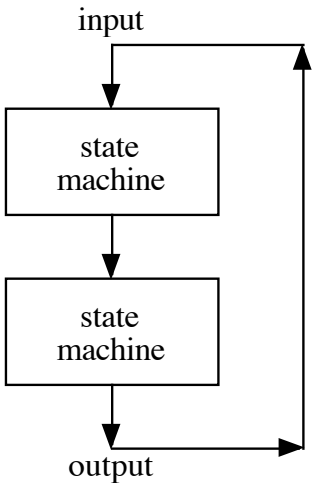

Figure 5.2: *The basic pipelined structure of many processor microarchitectures.* Many conventional processor microarchitectures can be fairly well represented with a state machine model of this type. Two pipeline stages are shown where each pipeline stage also has feedback from its own stage output making the whole machine a cascade of pipelined state machines.

simulators for those microarchitectures are designed to fairly closely follow that sort of pipelined operation. However, the model of Figure 5.2 does not properly represent the microarchitecture of the OpTiFlow machine. The OpTiFlow machine consists of more complicated individual machine components that exhibit relatively complicated and involved sequential state machines within themselves that are not correlated with any designated prior or subsequent clocked operation in other components. The primary component within the OpTiFlow microarchitecture that is most independent of and uncorrelated with the operation of other machine components is the issue station. This component alone has within it many independent and somewhat loosely cooperating state machines that do not form a familiar pipelined state machine model as is often the case with conventional microarchitectures. Further, there are no simple pipeline oriented relationships between the issue stations and other components of the microarchitecture.

The nature of the state-transition flow within the OpTiFlow microarchitecture might best be described as being a distributed, loosely connected, but cooperating set of independent finite state machines. An attempt to simply depict this sort of arrangement is shown in Figure 5.3. The figure shows an example of multiple finite state machines (three are shown in the figure but many more than this are typical) with some degree of cohesion within themselves (their outputs feed back to their own inputs) but also with a degree of coupling of outputs from each FSM to the inputs of each of the others. It can be noted, to borrow some ideas from object oriented programming, that most hardware machine components within the OpTiFlow microarchitecture exhibit high cohesion and

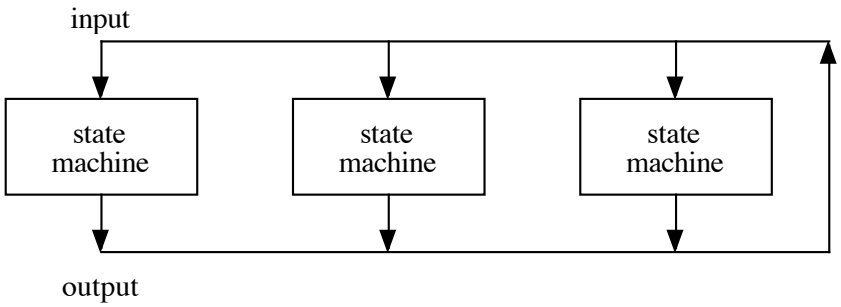

Figure 5.3: *An idealized state machine model more closely resembling that of the OpTiFlow microarchitecture.* Shown are multiple finite state machines where the outputs of each also form some of the inputs of each of the others. This arrangement of finite state machines more closely resembles the state and signal coupling encountered in a machine such as OpTiFlow.

low coupling. Just as object oriented programming serves to form a basis for modularization for large software programs, likewise the same principles can be applied to large finite state machines such as the logic for a processor. Although this is a good trait to have from the point of view of also modularizing the simulator (where each component becomes an independent module of its own), it represents problems for many microarchitectural simulators that were oriented towards modeling a machine best represented by the state model of Figure 5.2 rather than that of OpTiFlow, as idealized in Figure 5.3.

In the next section, we briefly further discuss why several of the more popular microarchitecture simulators are poorly suited to model the OpTiFlow microarchitecture.

5.2 Other simulation frameworks

In this section we briefly discuss some existing microarchitectural simulation frameworks and evaluate their suitability for our requirements in simulating OpTiFlow. These are also simulator frameworks that we actually evaluated and attempted to adapt for our purposes. These simulation frameworks were: ATOM, MINT, SimOS, and Simplescalar. These are each discussed in the following subsections.

5.2.1 ATOM

In 1994 Eustace and Srivastava, of the Digital Western Research Laboratory (then a part of Digital Equipment Corporation before it merged with Compaq), developed a very nice tool that allowed for a large number of architectural and microarchitectural research studies. Their tool was named ATOM [28, 29, 134]. It took as input an existing binary program (statically linked) and some

user created instrumentation and analysis subroutines and produced a new binary program as the tool's output. The resulting binary program, when executed, could both properly execute (completely faithfully) the equivalent of the original program while also allowing for the calling out to user inserted analysis subroutines as triggered by inserted instrumentation points in the program. The user supplied instrumentation subroutines would specify how the original program should be modified to trigger the calling of analysis subroutines. Both the instrumentation and analysis subroutines were written in the C programming language. With the additional of analysis subroutines being triggered from architectural events in the original program during its execution, this tool effectively created a simple execution-driven simulation environment. The fact that both the instrumentation and analysis subroutines could be written in the C programming language along with the relatively simple application programming interface for these subroutines made this tool very easy to use and attractive for a large number of research studies where elaborate microarchitectural detail was not needed.

Unfortunately, the fact that the tool essentially just executed the original program with call-outs to analysis subroutines at architectural trigger points also made the tool very difficult and cumbersome to use for more elaborate microarchitectural studies. For those studies where either wrong-path execution was needed to be examined or where a variety of concurrent microarchitectural components needed to be modeled alongside the original program's architected instructions, the tool was not well suited. This is actually something of an understatement since the tool was really only oriented towards calling analysis subroutines in response to architectural events as seen by the original program execution. Essentially this limitation of the tool makes it largely entirely unsuitable for the type of detailed microarchitectural simulation needed for the OpTiFlow microarchitecture.

5.2.2 MINT

Although several researchers and research groups have tried to make use of the MINT [157, 158] simulator for various microarchitectural investigations, this simulator was originally designed and developed for use in studying multiprocessor shared-memory synchronization techniques. Interestingly, and quite advanced for its time, the MINT simulator could simultaneously simulate the execution of multiple threads within a single process. This was indeed a very substantial accomplishment for this simulator, but this feature was also all but required since the intent of the simulator was to provide for the study and research of shared-memory synchronization mechanisms among multiple cooperating program threads of control. For its intended purpose, this simulator was very well suited and performed quite well. However, for our purposes, several design elements of this simulator were not well suited for simulating the microarchitectural dynamics of OpTiFlow.

One of the goals of this simulator was to execute the target program as quickly as possible while retaining the shared-memory interactions present in the program semantics. However, these twin goals allowed the simulator to take certain short-cuts in the simulation of the target program as long as these short-cuts didn't effect the shared-memory semantics of the original program. These simulation short-cuts were not suited for the detailed microarchitectural simulation that we needed to perform. Some of the liberties that the MINT simulator took with the simulated execution of programs are now briefly discussed.

Firstly, the simulator did not accurately execute the actual target binary program that was presented to it. Although the MINT simulator could read in a binary target program and execute it correctly [1] it actually modified the instructions of the target program during the process of loading it into simulated memory. Certain instructions were binary translated into others in an attempt to speed up the simulated execution of the program. Since the precise number and nature of the program instructions being executed were not of primary concern to the researchers using MINT, this was a laudable feature of the simulator since it yielded faster results while not interfering with the shared-memory program semantics. But this sort of liberty defeats some of the goals of a detailed microarchitectural study, such as what we wanted for OpTiFlow. With the simulation of OpTiFlow, not only would we like the simulator to actually simulate the execution of each instruction from the binary program (after all, we are trying to conduct research into instruction level parallelism and dynamic instruction scheduling), we also wanted to properly simulate and account for the various operand transfers among the issue stations and other microarchitectural components within our microarchitecture.

Another area where the MINT simulator took some liberties in the way that it functioned was to catch and stub out most all library subroutines that deal with the allocation and deallocation of heap memory. Subroutines of this class included the whole of the `MALLOC` and `FREE` family of library calls. Since the allocation and deallocation of heap memory did not involve the shared-memory semantics of the original program (no shared-memory was allowed to exist using heap-region memory), this was considered a feature. However, for our purposes, a good bit of interesting behavior of the target benchmark program being executed, with respect to instruction level parallelism, often exists in several of the system library subroutines, and the family of `MALLOC` and `FREE` subroutines are some of these.

Another area where the MINT simulator took some liberties with the simulated execution of target programs was with respect to the memory stack operations of the program during subroutine call and return instruction sequences. The MINT simulator did not simply execute through subroutine calls and returns as one might think (as happened on actual processors). Rather it trapped

[1] Actually, this simulator (MINT) has numerous errors in it that prevent it from executing, or correctly executing, many of the programs from the SpecINT-2000 benchmark suite.

those instructions that were involved in calls and returns and maintained a separate call-return stack different from what would have been a part of the program memory space if it had actually been executed on a real processor. It is not entirely clear why the simulator designers made this sort of optimization (if one can call it that), but this removes much of some interesting program behavior and execution dynamics that we want to capture in our microarchitectural studies. Specifically, all references to the program memory stack are removed from the program's memory access trace for subroutine call and return instruction sequences. Because of this, we also lose the memory values loaded and stored at locations of the target program's stack at the occurrence of subroutine call-return events.

Besides the deficiencies already mentioned, the MINT simulator was not well designed as a program itself and this alone substantially limits its usefulness for new and aggressive microarchitectural studies. The primary problem is that it is too difficult to hook new microarchitectural dynamics into the simulator since the simulator was not designed in a modular fashion to easily allow for that. The simulator was oriented towards fast simulation of the target program, for which it performed well, but it was not suited to modifications that deviated from this objective. Although the simulator allowed for the registration and insertion of call-back subroutines for the monitoring of certain architectural or microarchitectural events, this event registration structure was not fine-grained enough to be suitable or useful for our purposes.

Further, the MINT framework did not include any existing code or code modules for the simulation of machine microarchitectural components. Any such microarchitectural component simulation would have to be developed and spliced (integrated) into the existing MINT system. This would not have been an easy task given the poor code organization and low existing modularity of the MINT simulator. For these and other reasons, MINT was not a good choice for adaptation to our simulation needs.

5.2.3 SimOS

The SimOS simulator [48, 108, 109, 168] is an exceedingly nice research platform for conducting architectural or certain microarchitectural studies where a whole operating system needs to be included into the simulated execution mix. Indeed this is what this simulator framework was primarily designed and developed for. Other microarchitectural simulation frameworks do not allow for the complete simulated execution of a whole operating system, along with all necessary system-level and user-level library software for the proper execution of application programs. The problem for simulator developers is that creating a real life-like machine, along with enough of its peripherals, so that a complete existing operation system can be executed on it is too large of a task for most research organizations. Further, operating systems are very exacting in that

they require not only a precise machine simulation, but also a machine along with essentially all of the expected supplied peripherals that generally come with am actual commercial machine. Many of these peripherals may not be obvious since they are often things like: timers, power-control hardware, fault handling registers, non-volatile memory store (for storing time-of-day and other persistent configuration parameters across machine reboots), console hardware, and certain diagnostic hardware and registers. Simulating an actual commercial machine to the point where an associated operating system for that machine can be booted up on the simulator represents a unique research simulation space. The SimOS simulator succeeded in addressing this simulation space. This simulator is able to model a base machine sufficiently well such that a slightly modified version of the Silicon Graphics Incorporated (SGI) Irix [2] operating system is able to boot up on the simulated machine and continue on to execute system and user programs just as it would have in an actual machine environment. This represented a very significant accomplishment for a research simulator.

Although the goal for SimOS was an important one (simulating the execution of a whole operating system), some tradeoffs were made in the achievement of that goal. Obviously the speed of the execution was an important consideration. When executing an entire OS along with its many processes, attention to simulation speed is a very important consideration. Program fragments of an operating system (or system-level processes for that matter) cannot be executed instead of a complete program as can be done in many other simulation frameworks. To address the issue of execution speed, SimOS allows for the replacement of the core instruction execution module. Although at the time of our evaluation several instruction execution modules were available or in development, none of them were suitable for our purposes. Further, the time investment needed to fully integrate the detailed microarchitectural simulation features that we needed into the SimOS framework appeared to be too great. Other simulation alternatives appeared to provide a more timely avenue for our work.

It should also be noted, that for our present work, the need to simulate an entire operating system along with its subordinate programs was not one of our requirements. Although many researchers specifically want to see the behavior of the operating system itself executing on the simulated hardware, this was by no means a significant concern for us. Rather, those researchers who are trying to capture the behavior of complex workloads (such as transaction processing, database, or web serving) are those who most benefit from having the operating system simulated along with everything else. For our research work, being at the preliminary stage that it is at, it is quite entirely sufficient to sequentially execute several individual programs of a benchmark suite

[2]The SGI Irix operating system is a Unix-flavored system with most of the features of AT&T Unix System V Release 3 and AT&T Unix System V Release 4, as well as being substantially POSIX compliant. `UNIX` is a registered trademark of X/Open.

rather than having them time-sliced against each other under the control of an operating system. This has been the more historic approach to microarchitectural simulation, and is generally entirely appropriate for very novel research ideas like those of our present work: Resource Flow computing and the OpTiFlow microarchitecture. Moreover, the computing environment and resources needed for supporting the SimOS system is not as simple as one might like. Rather its computing requirements are more suited to a larger research project where computing resources can be more easily amortized over a larger number of researchers. Finally, like the MINT simulator previously discussed, there were no existing microarchitectural framework modules available for adaptation to our purposes. For these and a few other reasons, this simulation system was not very appropriate for our use.

5.2.4 Simplescalar

The existing simulation framework that is by far the most used and most appropriate for architectural and microarchitectural research purposes is that of the Simplescalar simulation suite [6]. Recent polls have shown that approximately 40 to 50 percent of all microarchitectural research is conducted by using or adapting one of more of the Simplescalar simulators. The Simplescalar framework actually comes with a variety of ready made simulators. Some of these are more suited for studying branch prediction or cache and memory hierarchies, but two of the supplied simulators (in the version 4.0 beta distribution) provide much more detailed simulation of two specific microarchitectures. These two latter simulators are named Sim-Outorder and Sim-MASE respectively. The expectation is that for most researchers, they can start with one of the supplied simulators and adapt it for their purposes. Likewise, we also investigated how we might adapt at least one of these two latter microarchitectural simulators for our purposes. One of these simulators features a microarchitecture that employs a Register Update Unit (RUU) [132], while the other one (Sim-MASE) is substantially more detailed and instead employs a reorder buffer (ROB) [128] in its microarchitecture.

Although both of the latter two simulators are suitable for much microarchitectural research by the community at large, they are not suitable for all microarchitectural research. Even though the second of the two simulators (Sim-MASE) is substantially more detailed than the other one (Sim-Outorder), both of them are still quite simple as compared with how an actual machine executes at the register transfer level of abstraction. These simulators are coded in the C programming language and the core execution loop for each is very closely approximated by the pseudo-code shown in Figure 5.4. Upon examining the code shown in Figure 5.4, it can be seen that the whole of the state machine making up the simulated microarchitecture is very much pipeline-oriented. Specifically, pipeline stages consisting of:

```
void mainloop()
{

        while (not_done) {
                stage_commit() ;
                stage_execute() ;
                stage_issue() ;
                stage_dispatch() ;
                stage_decode() ;
                stage_fetch() ;
        } /* end while */

return ;
}
/* end subroutine (mainloop) */
```

Figure 5.4: *The core program loop in the Simplescalar microarchitectural simulators.* Shown is a very simplified microarchitecture schematic showing the instruction queue, architected register file, five function units, and a register update unit (consisting of four entries).

- fetch
- decode
- dispatch
- issue
- execute
- commit

can be readily seen. In a real machine, the output signals from the `FETCH` stage would normally flow to the input signals of the `DECODE` stage; and the outputs of the `DECODE` stage would then flow to the inputs of the `DISPATCH` state, and so forth. However, in this simulator framework, note that each of the above machine pipeline stages are called and executed in the reverse order as compared with the actual pipelined flow of signals among them. The reason for this is that the simulator designers use a cleaver trick to simplify the overall simulator program. Rather than each machine stage calculating next output state that would become that stage's present output state after the next clock transition (and input to its following stage), each stage instead calculates its next output state and overwrites the appropriate present state of the stage that logically comes after it in the pipeline. But since the code for the pipeline stages are all called in reverse order, this overwriting of present state does not present a problem since the stage getting overwritten had already performed its necessary combinatorial logic of its own present state. This scheme works

well for simple microarchitectures (like most all conventional microarchitectures) that follow a very strict pipelined structure. However, this scheme does not work well at all for a microarchitecture such as OpTiFlow since no clear pipeline structure is present among its many independent state machines.

Again, a reference back to Figure 5.3 more appropriately shows the situation in the OpTiFlow microarchitecture. In OpTiFlow, each stage may produce outputs that are inputs for several other stages where the stages do not have a single natural pipelined signal flow among them. For this reason, the state transition programming scheme employed in the Simplescalar simulator is entirely inappropriate (obviously causes chaos if accidentally tried). Trying to fit the OpTiFlow model into the simple Simplescalar programming pipeline-stage framework, besides being quite unwieldly, would also likely introduce numerous subtle bugs into the simulator. Many of bugs might be caused by accidentally (and incorrectly) overwriting the current state of a component before that same component got an opportunity to calculate its next state from its current state. Rather, for OpTiFlow a more traditional approach that is similar to what is often used for hardware simulation is more appropriate for the modeling of its more complex signal connectivity. This new and different simulation approach not only more closely matches the component cohesion and coupling attributes of our OpTiFlow microarchitecture, but it also served to minimize errors during the coding process of our new simulator.

Finally, if the reader suspects (through inspection of the code fragment shown in Figure 5.4) that extensive use of global variables were employed in the design of the Simplescalar simulator, that suspicion happens to be entirely correct. Although there is a sense in which some code modularity was employed within the Simplescalar framework (at least in some respects), the extensive use of global variables makes the code difficult to analyse from reading the code alone. Further, generally the larger a whole program is (being usually coded within many compilation modules), the more difficulties arise from the use of global variables. This situation exacerbates the existing difficulties of modifying or enhancing the existing code in order to meet new needs, not to mention adding to the difficulty of adding new code. This is another reason why the Simplescalar framework is somewhat better suited to smaller microarchitectural simulators rather than to large, more complicated, models (like OpTiFlow).

Our approach to a simulation framework attempts to minimize the above difficulties that were inherent in the Simplescalar framework, as well as with regard to the other frameworks previously presented. Our simulation framework is presented in the next section.

5.3 New simulation framework

In this section we introduce the basic framework for simulating the OpTiFlow microarchitecture. We do not present all of the details of our simulation framework here but rather only present a very simplified overview of its basic layout and operation. This will well serve our purposes for the present discussion. As already mentioned, due to our desire to model the OpTiFlow microarchitecture at the register transfer level, a simulation model is needed that will properly represent the calculation of next state while minimizing bugs and errors in the simulator due to the many independent state machines and their mutual coupling of signals from the outputs of some components to the inputs of others (again, reference Figure 5.3 for a simplified approximation of the state machine connectivity). Achieving this goal required a coding model that more explicitly represents both the current machine state as well as the calculation for the next machine state, within each component and thus the whole of the machine. Another goal for the simulator is to have the simulation code more closely match the cohesion and coupling characteristics of the actual hardware machine. From this latter goal, a third goal would also be very desirable. That is to make the simulator as modular as possible while observing the previous two goals already. A fourth goal is try to represent the hierarchical instantiation of hardware components and subcomponents through the simulator's coding framework. A fifth goal, that should not have to be stated but is done so here anyway, is that the simulator design should also aid where possible in minimizing coding errors. Fortunately, the first four goals above would seem to support the achievement of this last goal already. The goals for the simulator can be somewhat succinctly stated as:

- more explicitly represent the current state and the calculation of the next state for each component
- more closely match the natural cohesion and coupling of the hardware machine components
- be coded in as modular a fashion as possible
- represent the hierarchy of the machine hardware as much as possible
- and be coded in ways that minimize errors during design and programming

To achieve our simulator goals, we have adopted an object oriented programming model that also explicitly provides for the storage of the current state and next state within each object. Each object will also explicitly provide for the calculation of next state from the current state and current inputs. In our framework, each natural component of the machine will be represented and programmed as an object in the code. Each component will be responsible for its own operation internally as well as for the processing of those external inputs that go into its next state calculations.

```
/* IS object implementation in code */

class is {
        current_state ;
        next_state ;
        other_state ;
} ;

int is::init() {
}

int is::comb() {
        do_stuff ;
        calculate_next_state ;
}

int is::clock() {
        current_state = next_state ;
}

int is::free() {
}
```

Figure 5.5: *An example of the code for a machine component.* A code object is shown with its private state and its various methods. Each code object represents a machine component in the whole of the microarchitecture.

Each component also presents its current output state (using the Moore state machine model for the present discussion) to other components for use as their possible inputs. Each component will also be responsible for managing any subcomponents that may be private to it. For example, a register file object may represent individual registers through the use of subcomponents. In this sort of design choice, the code object representing the register file would be responsible for managing the subobjects that represent each of the individual registers. This is a simple example (that may be more simple than actually warrants) but it serves to illustrate the idea. In something reminiscent of the outermost loop in the Simplescalar framework (illustrated in Figure 5.4), likewise our simulation framework will have an outermost machine component and it will be represented by an outermost code object. However, our outermost machine component (object) will itself be repeatedly called by a relatively simple loop. We illustrate with code examples each of the above ideas in the following figures.

First, in Figure 5.5, we show a greatly simplified example pseudo-code implementation of a machine component, as it is represented by a code object. This example shows what most components in the machine look like when programmed. As might be observed with the example pseudo-code, the object implementation closely resembles that of a class definition in an object oriented programming language. We draw some syntax from the C++ programming language to show how objects

are defined and how methods are called. The particular component depicted in Figure 5.5 is in fact an issue station component, as described back in Chapter 3, and its corresponding class name is abbreviated with the letters `is` in the figure. Of course, this code fragment is exceedingly simplified as compared with the actual issue station component implementation. In reality, the issue state is the single most complicated component in Resource Flow machines, but the pseudo-code presented serves our purpose to show the basic elements of modeled machine components.

Each machine component stores its own current and next machine state within itself. These two sets of state respectively correspond to the current state on the output of the component's clocked registers and the next state for those registers that will take effect after the next clock transition. The `other_state` of the object is not seen outside of the object at all and is only used as temporary storage for logic values being computed within a single clock period. The use of this latter state is a matter of design, convenience, or programming preference for the object's implementation. The pseudo-code also shows the four required methods for each object. These required object methods are:

- init
- comb
- clock
- free

The `init` method allows for the initialization of the object, and corresponds to the construction (initialization) of the represented machine component in the simulator. This method should initialize any internal object state (including any memory allocations) and also call the corresponding `init` methods of any and all of its subcomponents, if there are any. This method is essentially the object's *constructor* as would be encountered in a more formal object oriented programming language. Similarly, but in reverse, the `free` method first should call all of the `free` methods of its subcomponents (if any) and then free up any memory allocations of the current object. Similarly to the previous `init` method (that method being the object constructor), this `free` method corresponds to the current object *destructor* in the more formal object oriented paradigm. The `comb` method (an abbreviation for *combinatorial calculation*) is called so that the current object can calculate its next state from its current state and current input. This method should also call any of the subcomponents of the current object, if any. Finally, the `clock` method is used by the simulator to transition from one clock to the next. This method is generally quite simple and first copies its next state to its current state, and then calls the corresponding `clock` methods of all of its subcomponents, if any. Other methods are possible but are not required by the general simulation framework. Other methods would be created for acquiring inputs from other components or

for providing inputs to other components. Although most signals exchanged between components originate from a component's state outputs, this is not a restriction of the simulator programming model itself. Care is always required in the programming of an object to realize the implications of exchanging signals other than from the registered state output. The pseudo-code shown in Figure 5.5 is about as minimal as is possible and it doesn't have any subcomponents that need any managing.

A more realistic machine component example is shown in Figure 5.6. In this example, the class name of the current component is abbreviated with `iw`. Two subcomponent classes are employed and therefore need to be managed by the current component. The subcomponent classes used are `is` and `fetch`. As shown in the class variable definitions, when the current component is instantiated (allocated and initialized), all of its subcomponents are also effectively instantiated with it. In our programming model, the idea of object instantiation is subdivided into object allocation and its subsequent construction (or initialization). This subdivision of the idea of object instantiation can be seen in the pseudo-code example though the separate actions of allocating initial space for the subobjects and their subsequent initialization through calls to the respective `init` methods of the subcomponents. Note that in this example (Figure 5.6) the `fetch` subobject (of which there is only one) is statically allocated when the current (enclosing) object is allocated, but that the `is` objects (of which there are an unspecified number, as per some supplied simulation configuration) are instead dynamically allocated. In reality, dynamic object allocation is more the rule rather than the exception since the simulator is designed to allow for a configured number of many of its machine components. Of course, for any subobjects that are dynamically allocated, they would require corresponding dynamic unallocation (freeing) within the current object's `free` method. Through this approach, no global variables ever need be used for modeling any machine components. As is widely known in Computer Science, the use of global variables generally introduces complicated or confusing data flow semantics into the program and often leads to programming errors, thus their use is often minimized or avoided completely.

As with the previous object example (Figure 5.5), all subcomponents need to be primarily managed within the current component. This is evident in the present example (Figure 5.6), within each of the its required methods: `init`, `free`, `comb`, and `clock`. Also note that since the `is` subcomponents are dynamic in number (being also dynamically allocated and freed), they are managed through the use of loops in the current object methods in order to call the corresponding methods of the subcomponents.

Finally, Figure 5.7 shows a simplified version of the top of the main execution loop in our new simulator program. Inspecting the code, we can see that just one machine component object is instantiated and repeatedly called. This component is the `machine` object. In turn, this object, somewhat resembling the example of Figure 5.6, further instantiates appropriate subcomponent

```
/* IW object implementation in code */

#include "is.h" /* subcomponent */
#include "fetch.h"       /* subcomponent */

class iw {
FETCH   fetch ;
        IS      is[] ;
        current_state ;
        next_state ;
        other_state ;
} ;

int iw:init{} {
      initialize_our_stuff ;
      for (i = 0 ; i < n ; i += 1) {
            is[i].init() ;
      }
      fetch.init() ;
}

int iw::comb() {
        for (i = 0 ; i < n ; i += 1) {
                is[i].comb() ;
        }
        fetch.comb() ;
}

int iw::clock() {
        for (i = 0 ; i < n ; i += 1) {
                is[i].clock() ;
        }
        fetch.clock() ;
}

int iw::free() {
        for (i = 0 ; i < n ; i += 1) {
                is[i].free() ;
        }
        fetch.free() ;
        free_our_stuff ;
}
```

Figure 5.6: *A more complicated code example code for a machine component.* A more interesting and slightly more complicated code example is given for the modeling of a machine component. In this example, the current component object employs the use of two subcomponent types. One of the subcomponent types is statically allocated while the other type (of which there are an indeterminate number) is dynamically allocated.

```
/* at the top of the simulator */

#include "machine.h"

machine.init() ;

clock = 0 ;
while ( ! done) {

        machine.comb() ;

        machine.clock() ;
        clock += 1 ;

}

machine.free() ;
```

Figure 5.7: *The main execution loop for the simulator program is shown.* The primary execution loop for the simulator is shown calling the top machine component object. The top machine component object is responsible for hierarchically having all machine component objects systematically executed.

objects of the machine. Those subcomponents further instantiate their own subcomponent objects, and so forth with these subcomponents until the whole of the machine is created. Of course, all machine component objects are substantially designed to represent the microarchitectural machine components of a target processor. Through the hierarchical calling of component object methods from the top down, all components get executed in a systematic way. Of particular note is the return of the simulator program's execution thread to the top of the simulator loop of Figure 5.7 between the calls to the `machine.comb` and `machine.clock` methods. This point is effectively a synchronization barrier (physically an execution barrier) for requiring all state machines within the target to calculate their next state before the clock transition. Likewise, another implicit synchronization point after the call to the `machine.clock` method provides for all state machine to transition their next state to their current state (on the clock transition) before each state machine starts to calculate its next state again in the subsequent clock period.

Finally, although the pseudo-code examples are simplified, there are no global variables employed throughout the hierarchical object representation of the target machine. This programming device (no global variables) substantially simplifies the task of understanding how the machine works when trying to analyse its operation through inspection of the individual object code modules. The minimization of global variables also minimizes programming errors through mistaken or incorrect understanding of subtle data flow paths through the code that global variables tend to create in large programs.

5.4 Simulator implementation characteristics

Due to one of our research objectives (machine architecture independence), any machine instruction set architecture could have been chosen for our simulator. However, we have given consideration to what machine ISA might be most expedient also for our use. With this consideration in mind, we chose for our simulated machine (with the underlying OpTiFlow microarchitecture) the Alpha ISA. [124, 68] This ISA is (and has been) generally a good choice for architecture and microarchitectural research due to the several tools that have been designed around it. Two of these tools are the ATOM binary instrumentation tool [28, 29] and the Simplescalar tool-set itself. [6]. Through some programming, both of these tools can be useful in evaluating Alpha program characteristics and behavior through execution (either actual in the case of Atom, or simulated in the case of Simplescalar). Additionally, correct program execution can be compared between one of more of the existing Alpha tools and that of our own new simulator. In this way, programming errors that affect the correct execution of the target binary program can be caught.

The adoption of the Alpha ISA also allowed us to make use of (or some reuse of) some program code from another simulator; namely, the the Simplescalar simulator framework. In addition to providing development and verification assistance, the Simplescalar framework can be used in more creative ways as well. Specifically, the Simplescalar framework provides for a systematized description of the semantics of the Alpha ISA (among other ISAs). This semantical description of the Alpha ISA can be used in the development of simulators other than one based on the existing Simplescalar microarchitecture models. We have done exactly this with our own new simulator. We borrowed the Alpha ISA description code and have adopted it for our use. The ISA description from Simplescalar itself is something of a pseudo-language and is implemented using the macro processing capabilities of the standard C-language compiler. Although the ISA description capability was not entirely well suited for our purposes, its value as a correct description of the Alpha ISA's semantics was nonetheless very valuable and made its adoption a worthy undertaking.

The ISA description language from Simplescalar is somewhat better suited to the implementation of a fast architectural simulator (like the Simplescalar simulators) rather than for use in our Resource Flow microarchitecture (like OpTiFlow). This is due to the language's restriction in that source operands for instruction execution (as fixed through the description language) must come from architecturally named sources (either register or memory). Although this is exactly what is expected of instruction execution at the machine's architectural level, it is not well suited for use in modeling instruction execution when the instruction operands do not come from architecturally named sources, but rather from entirely speculatively created and maintained sources (such as is the case in machines using the Resource Flow execution model, like OpTiFlow). This restriction in the ISA description language somewhat reveals a bias on the part of its original designers to

not treat an instruction operand at the level of a first-class entity in itself, as we do with Resource Flow execution.

A simple example here will likely help somewhat. The description language for an ADD instruction will describe the instruction semantics as taking the first source register designated by its architectural name (eg. R6) and add it to its second source register (eg. R7) and place the result into its output register (eg. R8). The difficulty lies in the fact that in Resource Flow machines, operands of instructions no longer carry any direct association with any architected names in the machine. Rather, they have all been renamed and are entirely speculative. Operands for an instruction are instead only recognized by their ordinal position within the semantical representation of the instruction (for example: input 1, input 2, output 1). This difference in simulator approaches represents something of a semantical coding gap and must be reconciled through some additional programming. Unfortunately, some loss of simulator performance results, but the adoption of a working and verified ISA description is well worth the tradeoff.

As for what operating system to model, we chose the True-64 operating system. This has been the typical choice for most all research that has also chosen the Alpha ISA for implementation. Binary benchmark programs are available that were compiled for the Alpha ISA and against the True-64 operating system interface.

We also have adopted the style of stubbing out system calls in the same way as several other simulators have done (including for example, Simplescalar). We trap on the occurrence of a SYSTRAP instruction in the target binary program and semantically replace its architectural effect with those of executing the corresponding system call. Note that all system library subroutines (like within `libc`) are still simulated normally (thus capturing their behavior). Although this is a relatively simple way to handle system calls, it is not the best approach since it is somewhat prone to obsolescence by the operating system vendor who provided the system library (including the calls to kernel subroutines). The OS vendor is only obliged to implement kernel calls at the semantical level of the description of certain subroutine calls to the system library (generally `libc`). This is not the same as what happens within the kernel proper after a SYSTRAP instruction is executed. Therefore a better and more robust approach towards handling kernel system calls is to trap on the calling of the corresponding subroutine entry point within the `libc` library itself, and to stub out that subroutine's equivalent architectural state changes. Fortunately, for the True-64 operating system (unlike many other operating systems), changes within the `libc` code for handling system calls has been fairly minimal across updates and different releases of the OS. This fact makes our choice of how to handle OS calls a very reasonable one.

This discussion also brings to light the question of dynamic symbol binding and dynamic loading of shared-objects. Although dynamic linking is almost the rule in all modern operating systems, its implementation is exceedingly non-trivial and error prone. For this reason, we adopt the strategy of

only executing statically linked target binary executable programs. This strategy is adopted almost without exception for simulators that do not also execute the entire operating system itself. For those simulators (like SimOS) that do execute the entire operating system, the issue of handling dynamically linked executables becomes moot, since the operating system does that already without any further involvement by the underlying machine simulator. Fortunately, for the nature of our present work, capturing the execution-time behavior of the dynamic binding and loading code is not very important.

These above design choices enable source code programs (of any source language) to be compiled to the Alpha ISA and targeted for execution on the True-64 operating system. Compiled programs will be statically linked with the vendor supplied True-64 application and system libraries and the resulting binary in the executable Common Object Format File (COFF) can be loaded into the simulator for simulated execution. The COFF type executable is the native choice for use on the True-64 operating system. Binary programs suitable for simulated execution will also be able to be executed natively on an actual Alpha processor when it is also running the True-64 operating system. This last ability of our simulator, to execute identical binary programs as could be executed on an actual vendor system (Compaq True-64 OS), demonstrates our research design objective of creating an ISA-independent microarchitecture, through maintaining binary machine compatibility with a legacy ISA (the Alpha ISA in the present case). As it turns out, we also happen to maintain OS compatibility with the True-64 OS, but this was and is not a research objective.

Some of the design choices for the simulator that we have presented in this section are summarized as:

- implementation of the Alpha ISA
- adoption of the Simplescalar Alpha ISA description
- adoption of the True-64 operating system environment for system calls
- catching kernel trap instructions for system call handling and stubbing them out
- only providing for the execution of statically linked executables

Through these simulator design choices we have minimized the need for ancillary work that is not directly related to our research, while also not significantly compromising the validity of our execution results. These design choices also minimized the possibility of introducing errors through the adoption of both an existing Alpha ISA semantical description and the choice of the True-64 operating system environment for target programs.

5.5 Summary

We have introduced our approach for a simulator suitable for implementing the OpTiFlow microarchitecture. This simulator will enable the characterization of the OpTiFlow microarchitecture by representing the key structures of the microarchitecture at the register transfer level of abstraction. The key microarchitectural components of the microarchitecture that are most important for characterizing the behavior and performance of the OpTiFlow microarchitecture are the: issue stations, function units, the register file component, the fetch unit, and the load-store-queue component. The simulator is structured in such a way where it will fairly closely correspond to the natural hierarchical nature of the targeted microarchitecture. Major components will be primarily behaviorally modeled by implementing each component as a software object (using object oriented programming approach). Modeled machine components will then be structurally connected together (objects will effectively exchange signals with each other through method calls). The requirement for components to be executed in certain precise orders (as in the case with some other simulators) is greatly relaxed through the choice of calculating and maintaining both a current machine state and a next machine state. The calculated next machine state becomes the new current machine upon the occurrence of the next simulated clock transition.

A basic implementation of the simulator program has been completed. Many counters have already been placed within the various code pieces in order to gather characterization and performance data. The key performance metric obtained will be the instructions executed per unit clock (IPC) period. Additionally, a large amount of information is also available that describes the operating behavior of several individual components. A large degree of configuration of the simulator is also available. Configurable parameters include the numbers of various machine components, their clock latencies, the numbers of interconnecting buses, and some in some cases different operating strategies for an individual component.

Chapter 6

OpTiFlow simulation results

In this chapter we present simulation results on the OpTiFlow microarchitecture described in Chapter 4. The simulator for this present work is based on the model described in Chapter 5 (the last chapter). Our goal with this work is two-fold. First, we perform simulations on the OpTiFlow microarchitecture to validate its correct operation. This actually also serves as a validation of the Resource Flow execution model presented in Chapter 3. Second, we want to evaluate the OpTiFlow machine for its performance potential and to see how it compares against an equivalently configured baseline superscalar machine. For a baseline comparison machine, we use the model of a conventional superscalar machine similar to the Silicon Graphics Incorporated (SGI) MIPS R10000. We simulate the conventional machine using the SimpleScalar simulation framework. [9, 6] This framework contains a set of simulators of varying complexity. For our purposes, we chose the most complex of the simulators available. This would be the MASE simulator provided by the framework. This simulator attempts to model the targeted machine (the MIPS R10000) as closely as possible.

Both our OpTiFlow machine and the conventional superscalar machine execute the Alpha instruction set. We provide several performance comparisons between the OpTiFlow machine and the conventional machine. Although the two machines both execute the same instruction set, we attempt to match as many characteristics of the two machines as possible that they have in common. Where the machines differ (primarily in their execution cores), we attempt to roughly match the number of hardware resources. The idea here is that the machine that performs best, while using the same amount of physical resources (silicon), is the better machine. Presumably, a machine with much greater hardware resources should generally outperform another with less hardware resources (although even some poorly designed machine can fail at even this), so we attempt to configure machines with approximately the same hardware resources. We assume that both machines operate at the same clock frequency and therefore their relative performance can be indicated by the

number instructions per clock (IPC) that each achieves on average over the benchmarks executed.

The remainder of the chapter is organized as follows. We first present our experimental setup. We then show our simulation results which are subdivided into several major experimental groups. The first experiment is to compare our OpTiFlow machine with the baseline conventional superscalar machine. This is done over a number of machine configurations where both the baseline superscalar and the OpTiFlow machine are configured equivalently. The next experiment is to show the number of instruction re-executions that are occurring in a variety of OpTiFlow machines, and to attempt to understand some of the internal operational dynamics of the OpTiFlow machine in light of the data. The last of the experimental groups explores the behavior of the OpTiFlow machine when using different strategies (or design policies) for managing or carrying out re-executions within the OpTiFlow machine, and report which performs better. We then summarize our results and conclusions in the last section.

6.1 Experimental setup

For all of our simulations (both of the OpTiFlow machine and the baseline conventional superscalar machine) we use a set of program benchmarks that represent the types of workload that we want to target. We chose to target a workload that is both serial and sequential in nature (not obviously parallelizable at all at the program level) as well as being general purpose. The SpecINT-2000 benchmarks serve this purpose very well. They were established to represent a typical general purpose work load. The following benchmark programs from the SpecINT-2000 suite were used: BZIP2, CRAFTY, EON, GCC, GZIP, PARSER, PERLBMK, TWOLF, VORTEX, and VPR. All programs were compiled for the Alpha instruction set architecture (ISA). The specific compilation environment is given in Table 6.1.

Table 6.1: *Compilation environment used for our benchmark programs.* The operating system, specific C-language compiler, and targeted processor (for the compilation) is given.

component	version
machine word size	64 bits
operating system	Digital UNIX V4.0F
C compiler	DEC C V5.9-008 on Digital UNIX V4.0 (Rev. 1229)
processor	21264

For all simulations, the initialization phases of all programs are skipped using a fast-forward mechanism. This allows the subsequent functional simulation (which takes the real bulk of simulation time) to operate on a more characteristic part of the benchmarks. In this present work, 300

million instructions are skipped for each benchmark program used. After the initial fast-forward operation, a phase of one million instructions are executed to warm up machine components that have longer state residency times. This currently includes the cache hierarchy (L1 and L2 caches) and the branch predictor. Finally, we execute on the main functional cycle simulator for the next 50 million instructions. Table 6.2 provides the Spec2000 reference inputs used (where more than a single one was available) for each of the benchmark programs in our simulations. For all simu-

Table 6.2: *Benchmark program Spec2000 reference inputs.* Given here are the Spec2000 reference inputs used for each benchmark program when more than one was available. No input is listed for those programs that only have a single Spec2000 reference input.

program	input
BZIP2	program
CRAFTY	
EON	
GCC	166
GZIP	program
PARSER	
PERLBMK	make
TWOLF	
VORTEX	one
VPR	route

lations, we used separate instruction and data L1 caches, but a unified L2 cache. The data caches use a write-back policy with a least recently used block replacement algorithm. The execution window is flushed of younger (speculative) instructions on a branch misprediction (the same as most conventional machines). Common machine configuration parameters for all simulations (either on OpTiFlow or the baseline machine used for comparison) are shown in Table 6.3. In that table, execution function units (FU) are abbreviated as `FU`. Additional machine configurations that are only used for the simulation of the OpTiFlow microarchitecture are given in Table 6.4.

For both the baseline conventional machine and for the OpTiFlow machine, the benchmark programs that were simulated are the exact same object files, and resulted from a single compilation (the parameters of which were given in Table 6.1 above). Also for both the conventional machine and for OpTiFlow, executed the same set of instructions (both skipping leading instructions and functionally simulating from that point afterwards). And finally, unless otherwise noted, both machines used the machine parameters given in Table 6.3. As already stated above, the baseline is modeled and simulated using the MASE functional simulator while the OpTiFlow machine is modeled and simulated using the functional simulator first described in Chapter 5.

Table 6.3: *General machine configuration parameters.* These machine parameters are used for all simulations unless otherwise specified. These parameters apply for both the OpTiFlow microarchitecture as well as the baseline conventional microarchitecture, unless otherwise specified.

parameter	value
L1 I cache size	32 KBytes
L1 I organization	direct mapped
L1 I block size	32 bytes
L1 I cache access latency	1 clock
L1 D cache size	128 KBytes
L1 D organization	2-way set assoc.
L1 D block size	32 bytes
L1 D cache access latency	2 clocks
L2 cache size	1 MBytes
L2 organization	4-way set assoc.
L2 block size	32 bytes
L2 cache access latency	20 clocks
main memory access latency	150 clocks
branch predictor	2-level w/ XOR
	16k PHT entries
	8 history bits
	32k BHT entries
	sat. 2-bit counter
fetch width	8 instructions
FU issue width	4 operations
number integer FUs	4
number other FUs	1 each
integer FU latency	1 clock
other FU latencies	3 to 18 clocks

Table 6.4: *Additional machine configuration parameters for OpTiFlow.* These machine parameters only apply to the OpTiFlow microarchitecture. There are not analogous parameters for the simulator used in the baseline conventional machine.

parameter	value
forwarding buses	8
bus traversal latency	1 clock

6.2 Simulation results

In this section we present the results of several experiments that evaluate the OpTiFlow machine. The first experiment is to compare the OpTiFlow machine against a baseline conventional superscalar. The next experiment compares the OpTiFlow machine against itself under differing configurations. Next we perform an experiment where we compare two different modes for handling re-executions within the OpTiFlow machine. Finally, we conclude by comparing the best performing OpTiFlow machine against the baseline conventional superscalar.

6.2.1 Comparison with a baseline conventional superscalar

In this section we present the results of a set of experiments where we compare the performance of the baseline conventional superscalar machine with that of our OpTiFlow machine. All of our benchmark programs (discussed previously) are executed on each machine and the IPC results compared. The two machines are compared under a variety of configurations. In all cases, a set of two machines to be compared are equivalently configured with the same number and type of hardware machine resources. Although the two machines (baseline and OpTiFlow) are organized very differently, they each still share the idea of having a definite number of instructions within the execution window (or *instructions window* as it is often called in the literature), and a definite number of execution resources available to execute instructions in parallel. For each performance comparison, the numbers of each of these types of resources are set to be the same. Unless otherwise

The details about the design and organizations of the OpTiFlow machine was previously presented in Chapter 4. More information about the baseline conventional superscalar machine, and how it is equivalently configured with a matching OpTiFlow machine is presented in the next section. The following section then gives the IPC performance results of each of the two machines over a set of different machine configurations.

The baseline machine

The baseline superscalar machine will be modeled by the SimpleScalar MASE simulator. This simulated machine model includes an instruction window consisting of reservation stations and a reorder buffer (ROB) to store speculative execution result registers pending commitment. Instructions for the baseline superscalar are fetched and dispatched to the instruction window where they wait for input data dependencies to become ready. When all instruction input dependencies are ready, and as issue bandwidth allows, instructions are issued to the function-unit pipelines. Register results are stored in the ROB until commitment. Both the baseline superscalar and our presented microarchitecture flush the execution window on a resolved mispredicted conditional branch. This is

a fairly typical superscalar execution arrangement and is fixed in the SimpleScalar MASE simulator. The MASE simulator does not model the particular bus and interconnection arrangements that a real machine would have and so it is likely not quite as conservative as our simulator model might be in that regard. We can model both a fixed number of buses and the consequential waiting by units for access to bus transfer resources.

Although an exact comparison of the two machines is not possible due to their very different construction, we have arranged for both the baseline machine and our machine to have either an exact or a very close correspondence in the type and number of hardware resources each possesses. Both machines are configured with identical cache arrangements and cache configurations. Both also employ the same branch predictor and predictor configuration. Both also implement a four-wide issue machine. For both machines, the issue width is the maximum number of instructions that can be issued to an execution unit in a single clock cycle. However the baseline superscalar employs reservation stations and an ROB while our microarchitecture uses our novel issue stations. We therefore roughly equate the number of issue stations of our OpTiFlow proposed machine with the combination of both the number of instruction window slots (reservation stations) and ROB entries of the baseline superscalar.

Something to note that is different between the baseline machine and the OpTiFlow machine is that the baseline machine does not model a limited access to either its instruction window, architected register file, or its re-order buffer. It is assumed by the MASE simulator modeling the baseline conventional machine that these structures can be accessed in parallel as much as might be needed in any given clock period. In comparison, the OpTiFlow machine is strictly limited in how many operations can be carried out in parallel (in any given clock period) on its analogous structures. More specifically, all operands flowing between the issue stations of the OpTiFlow microarchitecture (reference Chapter 4 as desired), the architected register file, and the load-store-queue (LSQ) share a common set of eight parallel buses (as was specified in Table 6.4. The result of having a limited number of buses available for operand transfers within the OpTiFlow machine means that desired accesses to these buses, by components desiring to transfer an operand to another part of the machine, have to stall when bus bandwidth is not available. Further, it should be noted that all bus transfers in the OpTiFlow machine take an additional one clock period, in additional to any additional arbitration clock periods and possible stalls waiting for bandwidth.

Performance comparison results

We have performed several experiments comparing the performance of the OpTiFlow microarchitecture with the equivalently configured baseline superscalar machine. In this set of experiments

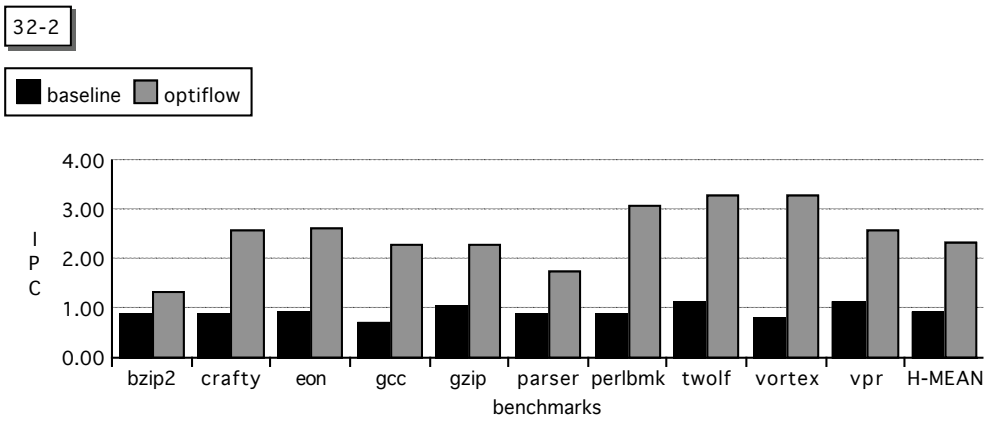

Figure 6.1: *Performance comparison of a configuration of the baseline with OpTiFlow.* The IPC performance of the baseline superscalar and OpTiFlow are shown. Both machines are configured for 32 instructions in-flight within the execution window and an issue width of two.

we explore six different machine configuration design points for the baseline and OpTiFlow machines over all of the benchmark programs. For convenience we will represent the configuration design points used in these experiments with a tuple of two numbers (a 2-tuple). The first number gives the maximum number of instructions residing within the execution window of the machine at any given time, and the second number will give the maximum instruction issue width. The maximum number of instructions within the execution window correspond to issue window slots or reservation stations for the baseline conventional machine, while they correspond to issue stations for the OpTiFlow machine. The maximum instruction issue represents the maximum number of instructions that can be executed simultaneously, and is the same for each machine. Figures 6.1 through 6.6 show the IPC performance results for each of the six machine configurations that we have explored. The benchmark named `H-MEAN` in the figures is the harmonic mean IPC across all benchmarks. The 2-tuple used to represent each design point is shown in the upper left corner of the figures, along with the legend for the data itself.

From the IPC performance results, we note that the OpTiFlow machines performed better than the corresponding baseline conventional machines for each configuration design point. We also note that each machine, whether the baseline conventional superscalar or the OpTiFlow, exhibits diminishing IPC performance returns when the issue width of the machine is increased. This trait is common among all machine microarchitectures and the OpTiFlow microarchitecture is not different in this regard.

We summarize the IPC performance results of this section by showing the harmonic mean IPC speedup of the OpTiFlow machine over that of the baseline conventional superscalar machine

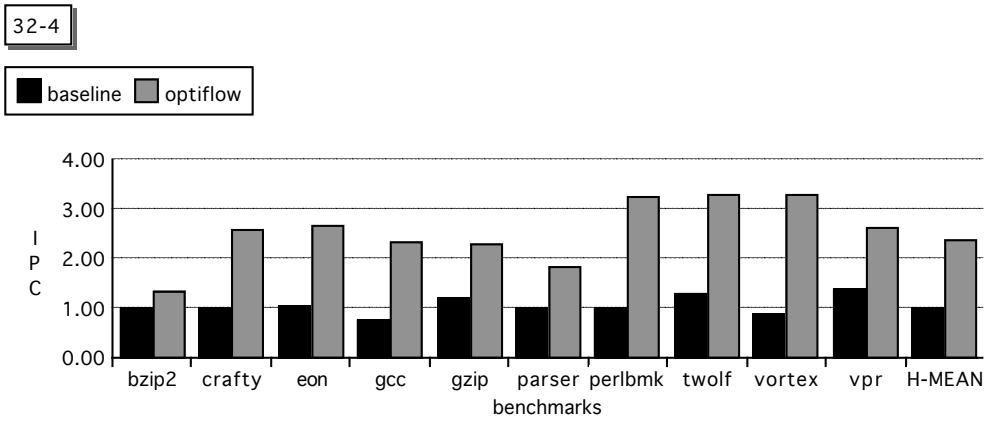

Figure 6.2: *Performance comparison of a configuration of the baseline with OpTiFlow.* The IPC performance of the baseline superscalar and OpTiFlow are shown. Both machines are configured for 32 instructions in-flight within the execution window and an issue width of four.

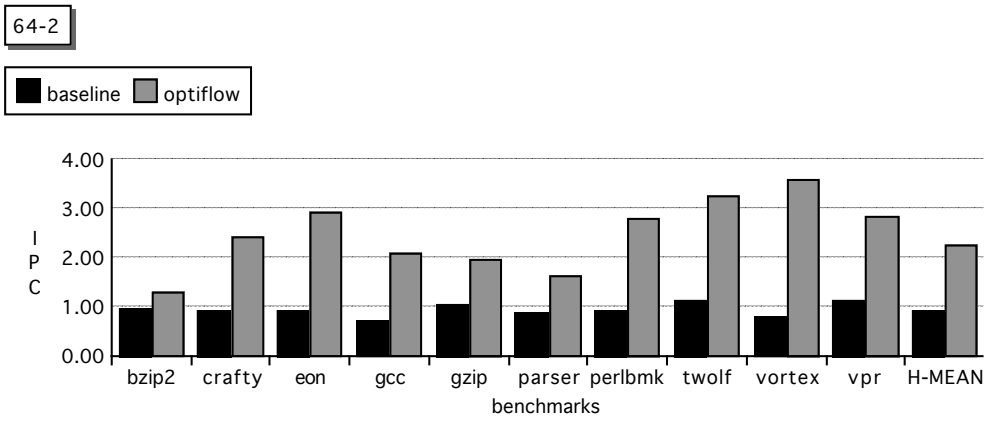

Figure 6.3: *Performance comparison of a configuration of the baseline with OpTiFlow.* The IPC performance of the baseline superscalar and OpTiFlow are shown. Both machines are configured for 64 instructions in-flight within the execution window and an issue width of two.

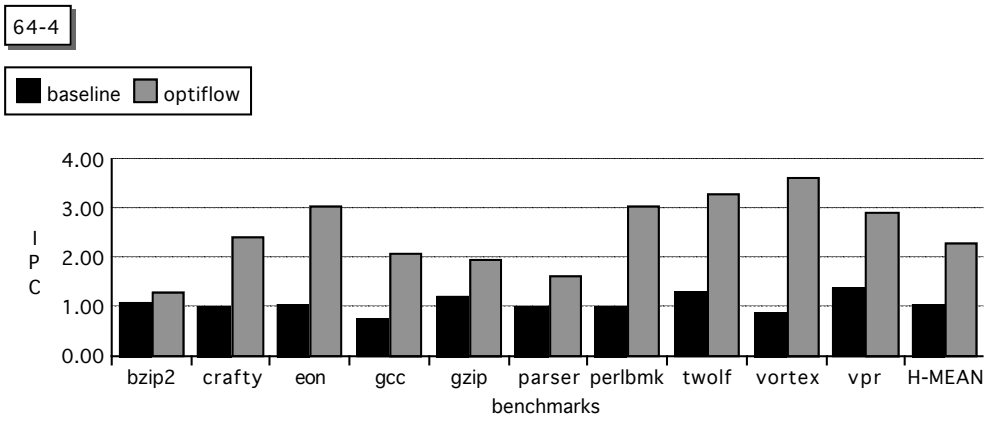

Figure 6.4: *Performance comparison of a configuration of the baseline with OpTiFlow.* The IPC performance of the baseline superscalar and OpTiFlow are shown. Both machines are configured for 64 instructions in-flight within the execution window and an issue width of four.

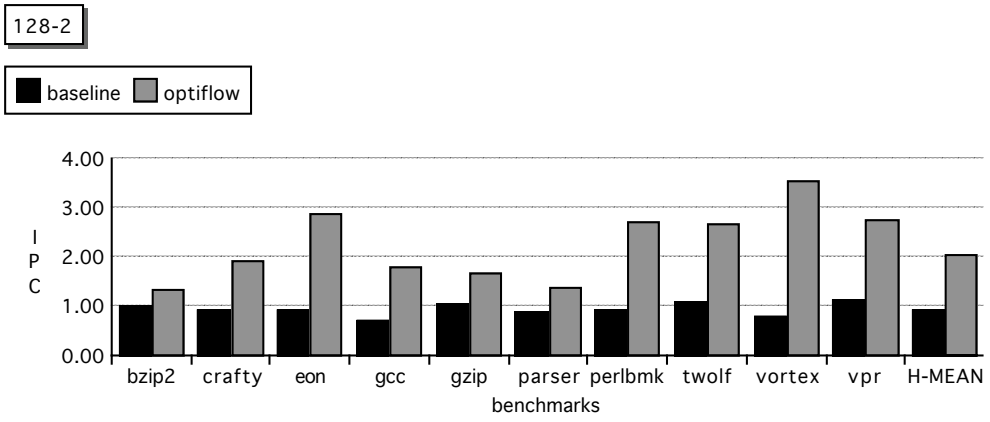

Figure 6.5: *Performance comparison of a configuration of the baseline with OpTiFlow.* The IPC performance of the baseline superscalar and OpTiFlow are shown. Both machines are configured for 128 instructions in-flight within the execution window and an issue width of two.

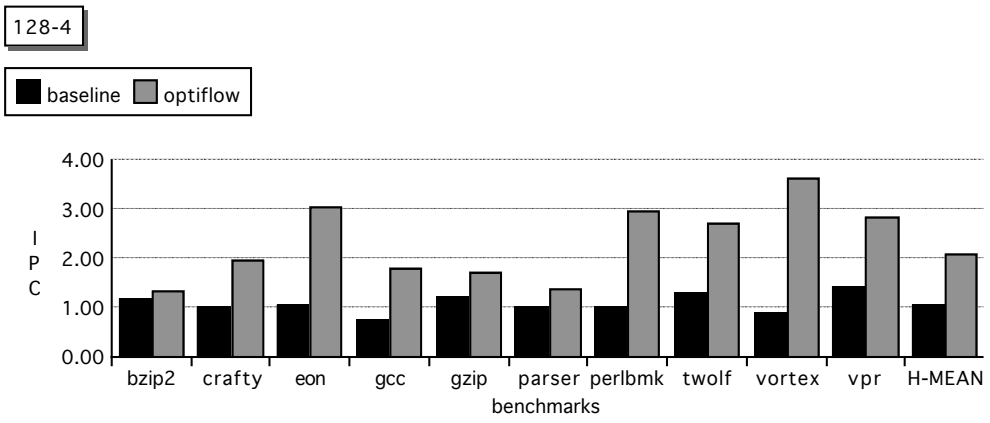

Figure 6.6: *Performance comparison of a configuration of the baseline with OpTiFlow.* The IPC performance of the baseline superscalar and OpTiFlow are shown. Both machines are configured for 128 instructions in-flight within the execution window and an issue width of four.

for all six experiments shown in Figures 6.1 through 6.6 as well as for the the `129-8` machine configuration in in Figure 6.7. This same information in numeric form is shown in Table 6.5. Each machine configuration is identified in the figure and the table by its 2-tuple representation. All IPC speedups are computed from the harmonic mean IPC over all benchmark programs simulated for each machine configuration.

From the IPC speedup information given in both Figure 6.7 and Table 6.5 we observe that the OpTiFlow machines perform at approximately twice or more (in terms of IPC performance) compared with the equivalently configured baseline superscalar machine. This is a fairly significant result when we consider that the same resources in terms of instructions within the execution window and the issue width of both the baseline and its corresponding OpTiFlow equivalent is the same. This shows that for the configured machines explored here that the novel microarchitectural organization of the OpTiFlow machine (based on the Resource Flow execution model) clearly outperforms that of the conventional superscalar machine. We also note that the speedup of the smaller issue-width OpTiFlow machines is greater than those of the larger issue-width machines. This latter aspect of the OpTiFlow performance is further addressed using the results of the next section.

6.2.2 Examining instruction re-execution

In this section we present data for the number of instruction re-executions that occur within the OpTiFlow microarchitecture as compared with the number of committed instructions executed for

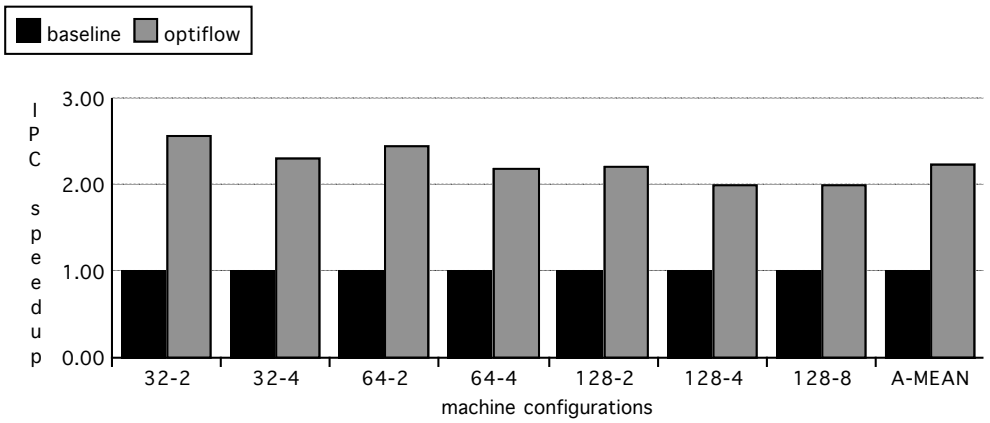

Figure 6.7: *The IPC speedups of variously configured OpTiFlow machines over their equivalent baseline machines are presented.* The IPC speedups of seven machine configurations are presented. The machine configurations are represented by their 2-tuples. The speedups shown are those of the OpTiFlow machines as compared with the baseline machines. The baseline machines therefore show an IPC speed of 1.0 and is provided for bar-height comparison purposes. All IPC speedups are computed from the harmonic mean IPC over all benchmark programs simulated for each machine configuration.

each benchmark program. This information is provided for a variety of OpTiFlow machine configurations. This number is best represented as a percentage of committed program instructions. The number of committed instructions that a program executes for a given program segment is the same as the architected number of instructions executed for that same program segment. This is the view of program execution from the perspective of the programmer. Additionally, and unknown to the programmer, the microarchitecture may actually execute instructions more than once as a consequence of internal microarchitectural factors. Some of these microarchitectural factors can include: flushing the execution window on branch mispredictions, flushing of instructions due to handling a page fault, and other related causes. These factors just described are common to both the OpTiFlow microarchitecture and conventional microarchitectures. Even conventional microarchitectures that do not perform speculative execution still need to periodically re-execute certain instructions when those instructions were the cause of either page faults or possibly if a machine interrupt occurred during the original execution of those instructions. However, the OpTiFlow microarchitecture performs instruction re-executions as a normal (not exceptional) matter of course as it executes all instructions. This is because of the way in which the OpTiFlow microarchitecture (based on the Resource Flow execution model) speculatively executes and re-executes instructions as it converges towards instruction commitment. As presented in Chapter 3 (Resource Flow) and Chapter 4 (OpTiFlow), the OpTiFlow microarchitecture speculatively executes instructions with

only feasible architected input operands in its attempt to both dynamically determine instruction input data dependencies as well as to eventually converge to instruction commitment. The instruction re-executions that the OpTiFlow microarchitecture performs in order to dynamically determine input operand dependencies are in addition to the various reasons for re-executions in conventional microarchitectures (the factors for re-execution already discussed above). For our purposes, we want to focus on the number of instruction re-executions that the OpTiFlow microarchitecture performs in addition to those that would occur in conventional machines. Since all machines (either OpTiFlow or something else) need to execute an instruction at least once in order to commit that instruction, we will use the number of committed program instructions as our base of comparison.

How many re-executions are occurring?

We present data for percentage re-executions for all benchmark programs simulated in eight different machine configurations. These data are presented in three figures broken out according to the number of issue stations in each group of configurations. Figures 6.8 through 6.10 contain this data. The benchmark named `A-MEAN` in the figures is the arithmetic mean percentage of instruction re-executions across all benchmarks. Figure 6.8 contains the data for two OpTiFlow machine configurations, each of which has 32 issue stations and issue widths of two and four respectively. Figure 6.9 contains the data for three OpTiFlow machine configurations, each of which has 64 issue stations and issue widths of two, four, and eight respectively. Finally, Figure 6.10 contains the data for three OpTiFlow machine configurations, each of which has 128 issue stations and issue widths of two, four, and eight respectively. The number of issue stations for each group of machine configurations is given in the upper left of each figure, while the issue width of the particular configuration is given by the legend for each data item graphed.

By examining the data in Figures 6.8 through 6.10 some general observations can be made. First something that is very interesting are the number of instruction re-executions that are taking place in these various machines. It must be remembered that all of these OpTiFlow machine configurations have an IPC performance better than an equivalently resourced conventional superscalar machine (reference the IPC comparison results in the previous section against the baseline conventional machine). Instruction re-execution percentages range from around 40% for the machines with 32 issue stations to around 80% for machines with 128 issue stations. However there are significant variations for re-execution percentages across all benchmark programs, and even more for those machine configurations with larger numbers of issue stations. These instruction re-executions are taking place in the OpTiFlow machine where pipeline bubbles are occurring in the baseline conventional machine. It is not intuitive that the percentage of instruction re-executions would be

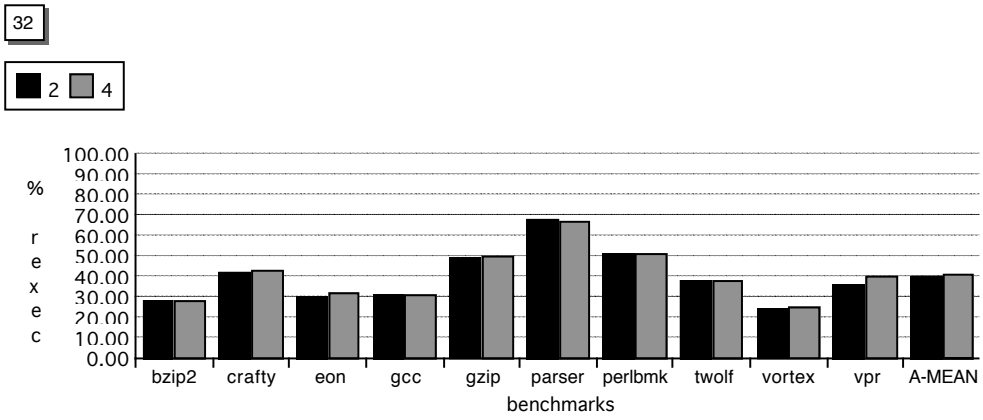

Figure 6.8: *Instruction re-executions expressed as a percentage of committed instructions.* The percentage of instruction re-executions is shown for two OpTiFlow machine configurations. Each has 32 issue stations, but issue widths of two and four respectively.

more than something on the order of 10% to 20%, but isn't the case. The reason for the larger than expected percentage re-executions is due to the very nature of the OpTiFlow microarchitecture and its Resource Flow execution philosophy. It should be remembered that instruction re-executions are not occurring selectively on program events that correspond in frequency to something like the frequency of conditional branches, memory operations, or instruction exceptions in the program instruction stream, but rather due to a combination of aggressive speculative instruction execution based on feasible input operands and the fact that mistakes are made in speculatively executing instructions before their proper input dependencies are determined. These conditions create the substantial amount of instruction re-execution that we observe. Further, just as IPC performance for a particular benchmark program is based on both the microarchitecture and the particular characteristics of that program, so too would we expect to see the percentage of instruction re-executions based (at least loosely) on both of these factors. This gives rise to the question of whether instruction re-execution percentages for particular programs are correlated to the IPC performance of those programs.

To try to better understand some general dynamics of the OpTiFlow microarchitecture, we will graph the percentage of instruction re-executions along with the associated IPC results for all of the benchmark programs simulated in three machine configurations. The three machine configurations that we choose (using the standard 2-tuple representation) are `32-4`, `64-4`, and `128-4`. As it turns out, these three machine configurations fairly well represent and show the general correlation with IPC that we want to explore. Figures 6.11 through 6.13 show, respectively for our three machine configurations, the juxtaposition of the percentage of instruction re-executions for each benchmark

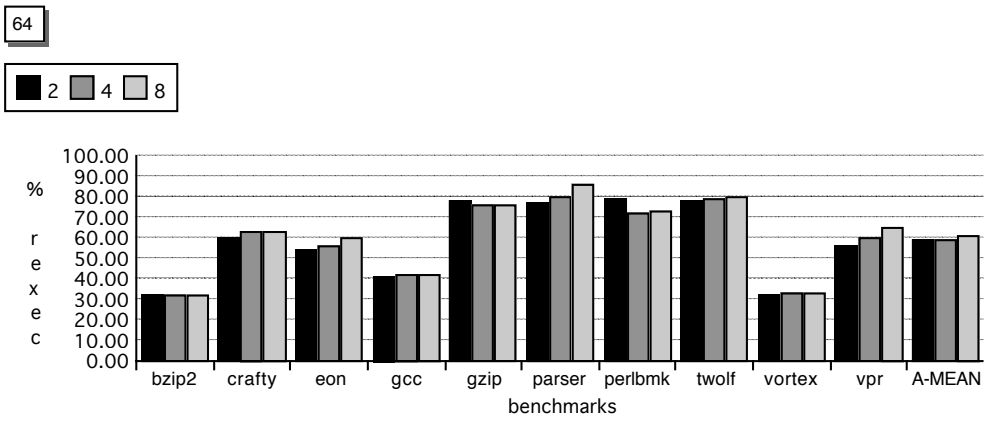

Figure 6.9: *Instruction re-executions expressed as a percentage of committed instructions.* The percentage of instruction re-executions is shown for three OpTiFlow machine configurations. Each has 64 issue stations, but issue widths of two, four, and eight respectively.

program with that same program's IPC performance. The benchmark named `A-MEAN` in the figures is the arithmetic mean percentage of instruction re-executions across all benchmarks. The machine configuration in each figure is show using the 2-tuple representation, and is shown in the upper left corner of each figure.

From Figures 6.11 through 6.13 a rough (but not high) correlation is evident between the percentage of instruction re-executions and the IPC performance of the benchmark program. These rough correlations continue from the small sized `32-4` up to the larger `128-4` machine. However two programs do not follow the correlation between percentage instruction re-executions and IPC at all (certainly not very well). These two programs are PARSER and VORTEX, and these will be discussed a little later on. The reason for the rough correlation of percentage instruction re-execution with IPC for most programs is due to the fact that these programs are not over utilizing the integer ALU function unit in the current machine configurations. The real correlation is the other way around than what we have first assumed. Programs that would normally achieve higher IPC performance on conventional machines will tend towards achieving higher IPC on the OpTiFlow machine. This is due, in part, to the internal instruction level parallelism (ILP) within the programs themselves. We have already seen from the initial performance comparison of the baseline superscalar with the OpTiFlow machine that the IPC performance of most benchmark programs on the OpTiFlow machine tends to track (with OpTiFlow performance being higher) that attained on the baseline conventional machine. This is evident in Figures 6.1 through 6.6 where we compared the IPC performance of the baseline conventional machine with that of OpTiFlow. The only notable exceptions to the trend being the GCC and VORTEX programs, and to a little

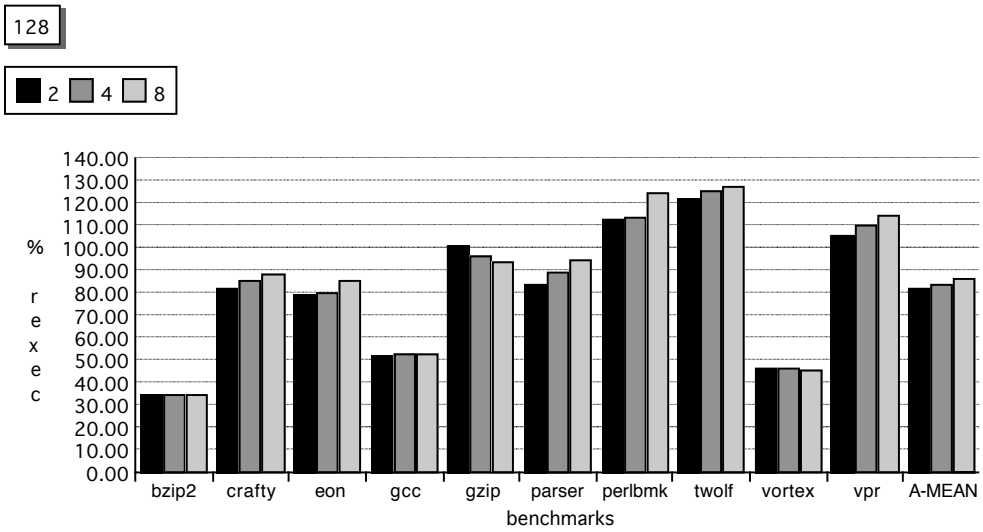

Figure 6.10: *Instruction re-executions expressed as a percentage of committed instructions.* The percentage of instruction re-executions is shown for three OpTiFlow machine configurations. Each has 128 issue stations, but issue widths of two, four, and eight respectively.

lesser extent, the PARSER program. Both of these programs tend to perform a little poorly on the baseline conventional machine, while they are much less constrained on the OpTiFlow machine (thereby performing better). In most cases, programs that achieve high IPC mean that they are executing more instructions per unit clock than other programs (the very definition of IPC). But this increased execution rate per clock also means that these programs are more susceptible to false operands (operands that will not be committed) causing unnecessary and wasteful executions. As long as these extra re-executions do not cause the program to over utilize the total available hardware execution resources, the program continues to perform well even though its percentage of instruction re-executions also increases. It is when the number of re-executions starts to saturate the available execution resources, that performance degrades. This trend is already partially evident with the IPC results from OpTiFlow that we have already seen. A program that has already started to saturate the available execution resources due to instruction re-executions is the PARSER program (and to a lesser extent the GCC program). An analysis of the PARSER program below will help to illustrate this problem.

The VORTEX program performs fairly well across most OpTiFlow machines for one main reason with two factors affecting it. The main reason that VORTEX performs well is that it is not saturating the integer execution function unit in almost all configurations. The percentage of instructions in VORTEX that require the integer ALU FU is only approximately 40% of its

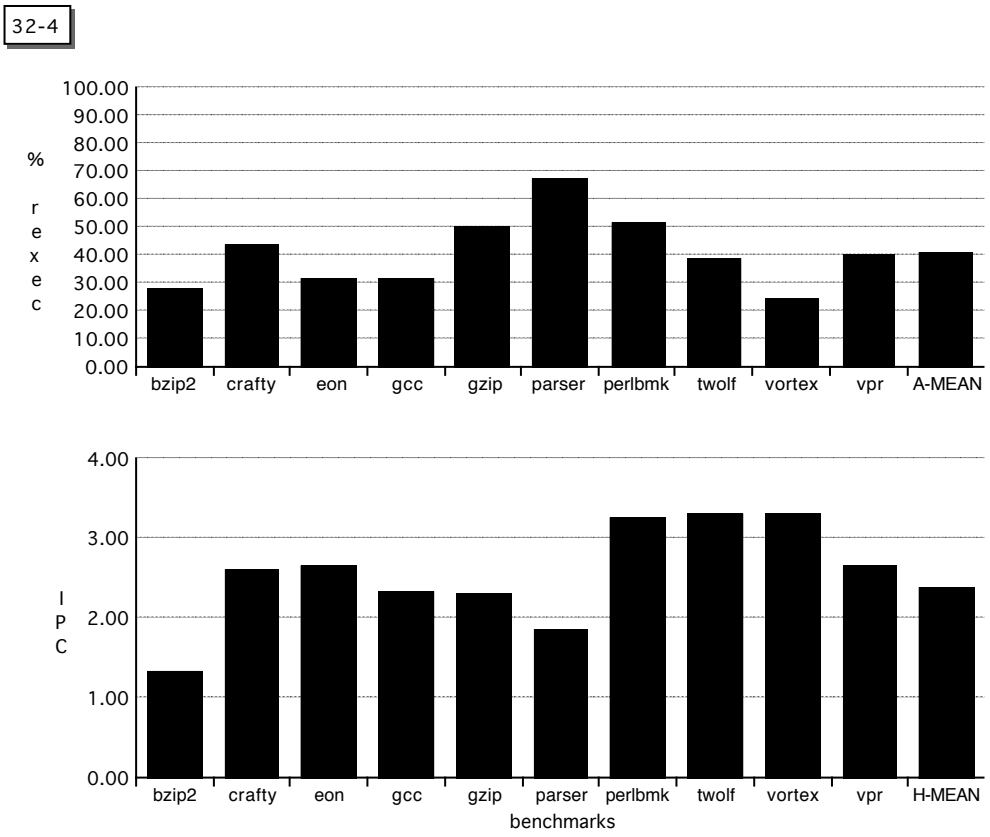

Figure 6.11: *The juxtaposition of instruction re-executions with IPC.* The top graph shows the percent instruction re-executions for the 32-4 OpTiFlow machine configuration. This is juxtapositioned with the graph of the same machine configuration below it.

committed instructions. The remainder of its instructions are divided into three main classes. These are memory loads and stores, and those simple ALU-type instructions that can execute within the issue station itself. About 27% of the instructions in VORTEX are memory loads, about 18% percent are stores, and the remaining 16% are very simple ALU-type instructions. All memory loads and stores execute within the issue station (partly due to the implementation). Helping matters is a good instruction and data cache hit rate. The instructions cache is returning results in an average of 4.2 clock cycles with a standard deviation of only 7.5 clocks (maximum clock latency is 301). The data cache (with about 27% of instructions going there) is performing even better with an average latency for results of 2.2 clocks with a standard deviation of 4.4 (and a maximum latency of 298 clocks). The VORTEX program is an object oriented database access program and it appears to be making good use of the data cache in particular. Further, the fact

that both memory instructions and the simple ALU-type instructions are executed within the issue station (the simple ALU-type instructions in a single clock) leaves the integer ALU FU available and under utilized. Grants for the integer ALU FU are returned immediately in approximately 98% of the time on the `32-4` machine configurations and about 95% of the time on the `128-4` machine configuration. The VORTEX program also has a fairly low amount of instruction re-executions compared with the other benchmarks. This both frees up the execution functions units (FU) to be more available when needed (which is certainly the case with this program as already mentioned), but also indicates that fewer false operands are causing unnecessary instruction re-executions.

The more interesting case is that of the PARSER program. It performs fairly poorly across all machine configurations compared with the other programs. It performs somewhat or fairly poorly for similar reasons that VORTEX performs well. It requires a greater percentage of its instructions to go out to function units than many other programs. Approximately 52% of its instructions require an integer ALU function unit. Further, about 26% of the time, a request for the integer ALU function unit (the most used) has to wait due to congestion of the unit. Also, about 25% of its instructions going to the data cache are memory load instructions and the average data cache latency is 4.1 clocks (higher than many other programs) but also with a larger standard deviation on data cache latency (towards longer latencies) of 15.5 clocks! Both the saturation of the available integer ALU function units as well as somewhat poorer memory performance serve to lower the IPC for the PARSER program, but it is the problem with over utilizing the FUs – and particularly the integer ALU FU (the most important for these integer programs) – that presents problems for the OpTiFlow microarchitecture when scaled to larger sizes. The relatively poor performance of the PARSER program is only starting to show this more general problem with the OpTiFlow microarchitecture. The problem of high instruction re-executions saturating the available execution resources is a general one and is now discussed in more detail.

More re-executions are not necessarily good

In addition to correlating individual program IPC performance with its instruction re-execution percentage, there is an even more striking correlation for the percentage of instruction re-executions that we have only alluded to until now (so that we could first discuss the less significant case of individual programs). Note how we presented the raw instruction percentage re-execution data in Figures 6.8 through 6.10. We presented the data grouped according to the number of issue stations in the machine configurations. From the total data it is observed that those OpTiFlow machines that have a larger number of issue stations is the most significant (and dramatic) determinant of the percentage of instruction re-executions that can be expected. In Figure 6.14 we have grouped the instruction re-execution percentages of three machine configurations together to better show this.

In this figure we show the instruction re-execution percentages of the `32-4`, `64-4`, and the `128-4` machine configurations to summarize and illustrate the point. The benchmark named `A-MEAN` in the figure is the arithmetic mean percent instruction re-executions across all benchmarks.

From the percentage instruction re-execution data presented in Figure 6.14 it can be more clearly seen that there is, on average, a significant increase in the percentage of instruction re-executions with the increasing numbers of issue stations from each machine configuration to the next. The number of issue stations jump from 32 to 64 and then to 128, while the percentage of instruction re-executions goes from (on average – reference the A-MEAN entry in the figure) about 41% to 60% and then to 83% respectively. In contrast, notice from Figures 6.8 through 6.10 that the percentage of instruction re-executions does not increase nearly so must when the issue width of the machine increases within a given number of issue stations. This is a fairly striking result and shows that it is the number of issue stations, and not the ratio of issue stations to issue width (as might be otherwise supposed), that drives the number of instruction re-executions up. What happens in the OpTiFlow microarchitecture is that as the number of issue stations increases, the number of false operands (operands that are not the eventual committed value) that get forwarded to subsequent instructions (instructions in future program-ordered time) also increases. In effect, the machine lets false operands corrupt the executions and re-executions of subsequent instructions causing a sort of re-execution thrashing within the machine. More corruption due to false operands leads to more re-executions, which in turn can lead to more false operand corruption. This situation can also be termed destructive interference of operand interactions. By this we mean that the re-execution thrashing due to false operand corruption leads to poorer performance rather than greater performance – a classic consequence of a thrashing-type of situation. Eventually the number of re-executions increases to the point where constructive execution progress of the machine is restricted by saturation (over utilization) of the available execution resources. This condition leads to decreased execution performance for machine configurations with large numbers of issue stations. Note that for the GZIP program in Figure 6.13 (showing the re-execution and IPC performance for the 128-4 machine configuration) it is has also reached saturation of the available execution resources and has decreased in performance (as PARSER has) from the previous machine configuration of `64-4`, shown in Figure 6.12. In fact most benchmark programs show a slight (or modest) decrease in performance from the machines with less issue stations to those with more. This was also evident in the summary harmonic mean IPC speedup data for the OpTiFlow machine presented in Figure 6.7. Note in that figure how the harmonic mean IPC speedups for OpTiFlow decrease for those machines with increasing numbers of issue stations.

These data show us a second reason why the OpTiFlow microarchitecture is not performance-scalable. The first reason that it isn't performance-scalable is because it is not physically scalable. This is the same reason that the baseline superscalar microarchitecture (and basically all existing

microarchitectures) isn't performance-scalable – because it is likewise not physically scalable. But the second reason why the OpTiFlow microarchitecture isn't performance-scalable is due to the onset of re-execution thrashing from false operand corruption in those machines with large numbers of issue stations.

Happily, there are solutions to both of the problems that the OpTiFlow microarchitecture faces. First, unlike conventional microarchitectures, the OpTiFlow microarchitecture can be physically scaled through the physical repetition of its operand transfer buses. This is possible due to the very loose coupling between issue stations – even allowing for the possible use of asynchronous clocks between them in addition to arbitrary unit-step clock slips. The subject of enhancing the OpTiFlow microarchitecture to be physically scalable is taken up in Chapters 8 through 10. The same means to physically scale the microarchitecture also serve to break up and mitigate the forwarding of false operands to subsequent instructions. This, in turn, mitigates the problem of the lack of performance-scalability due to destructive interference of false operand interactions and re-execution thrashing.

6.2.3 Instruction re-execution policies

In this section we perform an experiments where we compare two different ways to manage re-executions within the issue stations of the OpTiFlow machine. There are many subtle possibilities (or policies) for managing re-executions within the issue station and the results that were presented in the last section (shown in Figures 6.1 through 6.6) represent only one such choice. The original choice made in the design of the issue station within OpTiFlow was that if a new input operand (or more than one) arrived while a previous instruction issue (for execution) for the present issue station was still being executed, that a new instruction issue would be sought. Remember that when an issue station wants to perform an instruction issue, it must first arbitrate for and wait for an available execution issue slot to become available. This is necessary since, in general, there are many more issue stations designed (configured) into any implementation of an OpTiFlow machine than there are execution resources. Further, execution resources are typed according to which type of instruction execution function they perform (integer ALU, floating point add-subtract, integer multiply, floating point multiply, floating point divide, et cetera). For this reason, execution resources are more generally referred to as execution function units (as was presented in Chapter 4 on the OpTiFlow microarchitecture). Any request for an execution resource also has to wait for the particular type of execution resource (or execution function unit) to become available. When an execution function unit becomes available for executing a new instruction issue, it returns a grant to that issue station that won the arbitration process for its issue request. The requesting issue station then proceeds to make its new instruction issue to the proper execution function unit

(FU).

In the original design choice for handling instruction issue, if a new instruction issue was sought while an existing request for an instruction issue was still outstanding no new instruction issue request was made but rather the input operands associated with the first issue request were effectively overridden with the input operands associated with the second request. When a grant for the initial request finally came back to the issue station, the issue would be made to the appropriate FU with the updated input operands. Since only a single instruction issue was requested and made by the issue station, only a single execution result would return from the FU back to the issue station. New output operands would be forwarded as may be required (if the value of the output operands changed from their prior values at all). However, if an instruction issue request was made and already granted and if one or more new input operands arrived before the corresponding execution results returned from the appropriate FU, a new instruction issue request would be made for an execution resource (execution slot). In this way, two separate instruction executions from the same issue station (on the same instruction instance but with differing input operands) could be outstanding and be executed simultaneously. This could circumstance even even proceed to have additional simultaneous executions outstanding (more than just two). In those circumstances where more than a single instruction execution is outstanding, the issue station would eventually acquire the results for all of them in turn. As an instruction result is returned from the FU back to the issue station (creating new output operands), the issue station would forward those new output operands to subsequent issue stations (located in future program-ordered time) within the machine.

The present policy of handling re-executions has some possible performance disadvantages associated with it (although it is not as easy to predict the consequences as might seem evident at first). One possible performance disadvantage is that since a second instruction execution was initiated before the first one finished, the result of the first instruction execution is essentially wasted within the whole of the machine – being eventually overridden by the results of the second instruction execution. Not only was an additional instruction execution function unit slot (execution resource) used up where it might not have been otherwise (if only one of the instructions executions was allowed to proceed to actual execution), the results from the first instruction execution were forwarded to subsequent (in program-ordered time) instructions, thus causing them to execute or re-execute unnecessarily since they would again be forced to re-execute upon eventual receipt of the output operands from the second instruction execution of the originating issue station. The effects of these interactions are neither obvious not intuitive. Not to confuse matters further, but also remember that it could have been the case that the results of the first instruction execution were those that would eventually be the proper committed program result (when commitment is eventually determined) and that the effects of the output operands associated with the second

instruction execution actually served to temporarily corrupt executions throughout the remainder the machine (in future program-order) and not just the present instruction or its immediately dependent instructions. We refer to this sort of condition (when value corruption occurs) as a form of *re-execution thrashing*. It also represents a degradation in performance when it occurs. In any event, all of these operand forwards and instruction execution interactions do appear to waste machine resources in terms of both operand forwarding bus bandwidth and execution resources (FU execution slots). We term the existing re-execution policy *overlap*, since it allows for multiple overlapping simultaneous instruction executions from a single issue station. We now explore one alternative policy for handling instruction re-executions that might address some of these apparent performance disadvantages associated with the original policy.

The second policy for handling re-executions serves to reduce both the number of instruction executions as well as unnecessarily forwarded output operands and the associated consequences of that in terms of both operand forwarding bus bandwidth and subsequent re-executions. The new policy will be for the issue station to not request a new execution resource when it already has an instruction currently executing (had already received a grant from the associated FU), but rather to wait until the first outstanding instruction execution is completed before issuing a second one. All other aspects of the current policy would be retained. In other words, each issue station would only be allowed to have a single instruction execution outstanding as any given instance. It would not be restricted from performing multiple instruction executions serially, as is also done with the original policy. We term this new re-execution policy *serial*, since it only allow for serial (one after the previous) instruction executions from a single issue station.

In Figures 6.15 through 6.17 we present IPC performance results for these machine configurations. The benchmark named `H-MEAN` in the figures is the harmonic mean IPC across all benchmarks. Again, we represent the machine configurations using 2-tuples as we did in a previous section. The first number of the total number of instructions that can be within the execution window at one time (reservations station or issue stations), and the second number is the issue width of the machine. The three machine configurations used for these experiments are representative of a larger set of configurations that we experimented with. The machine configurations shown in the figures are `32-4`, `64-4`, and `128-4`. The machine configuration (using its 2-tuple) is shown in the upper left corner of each figure, along with the data legend. The data associated with the the original overlapping re-execution policy is labeled as `overlap` in the figures, and the data associated with the new re-execution policy is labeled as `serial`.

As can be seen from the figures, the performance for the serial re-execution policy is slightly better in general than that of the overlapping re-execution policy. However, for some benchmark programs, the original overlapping re-execution policy was still better. Examples where the original policy were better include the VPR benchmark in the `64-4` and `128-4` configurations, the EON

benchmark in the `128-4` configuration, and the PERLBMK and VORTEX benchmarks in the `128-4` configuration. However, in general there is a small but significant overall performance improvement when all benchmarks are considered together.

Table 6.6 summarizes the harmonic mean (taken over all benchmark programs) IPC speedup, in a percentage, of the serial re-execution policy over the overlapping re-execution policy for a variety of machine configurations (not all of which were shown in figures). From these results, we can conclude that both re-execution policies perform approximately the same (with the serial re-execution policy having a slight overall edge) so the decision about which to implement should be based on other considerations. One such consideration might be power consumption of the machine for each of the two policies. But it is not easy to predict which policy might actually be better in that regard, or even if other policies are better yet. Although the serial re-execution policy might appear to have a lower number of overall executions as opposed to the overlapping policy (possibly yielding fewer overall executions – and lower power consumption), this should be confirmed with experiment given the dynamic complexities of the operation of the OpTiFlow microarchitecture. This confirmation is beyond the scope of our present work.

But we can still examine the number of number of instruction re-executions each policy generates. To compare the percentage of instructions re-executions between the two -re-execution policies (`overlap` and `serial`) we examine three machine configurations using both policies. The three machine configurations that we will examine (being as they are fairly representative of the wider data set) are the `32-4`, `64-4`, and `128-4` configurations. Figures 6.18 through 6.20 show the percentage instruction re-executions for all benchmark programs and each re-execution policy over each of the three machine configurations chosen. The machine configurations are identified using their 2-tuple representation and are shown in the upper left corner of each figure. The benchmark named `A-MEAN` in the figures is the arithmetic mean percent instruction re-executions across all benchmarks.

From the data in Figures 6.18 through 6.20 it can be seen that the percentages of instruction re-executions do not vary significantly for most all benchmarks and machine configurations between the `overlap` and the `serial` policy. Although the results for the two policies are not identical, there is not a large substantial difference between the two. Referencing the A-MEAN pseudo benchmark (arithmetic mean over all benchmarks) in the figures is useful to see just how close each re-execution policy operates in terms of percentage re-executions. For reference, arithmetic mean (A-MEAN) percentage re-executions for the two re-execution policies over several (seven) machine configurations is provided numerically in table 6.7. These data (from Figures 6.18 through 6.20 and Table 6.7) do not help us in determining which policy might be best to implement, and so this is an open design issue for the OpTiFlow machine (or any Resource Flow based machine) design implementor.

6.3 Summary and conclusions

We had two goals with the simulation work of this chapter. First, we wanted to validate the operational correctness of the OpTiFlow microarchitecture (presented in Chapter 4). Second we wanted to perform some initial evaluation of the microarchitecture. We have achieved both of these goals with the present work. We have simulated the OpTiFlow microarchitecture using the simulator that was developed and presented in Chapter 5. That simulator was designed specifically to model the OpTiFlow microarchitecture and was necessitated due to the unusual subtleties and complexity of both OpTiFlow and any Resource Flow oriented microarchitecture. That simulator was designed to execute existing legacy binary programs compiled for the Alpha instruction set. Binary program compatibility with existing ISAs (an important feature in the marketplace) was maintained as an objective and that was achieved. With the present simulation work on OpTiFlow, we have validated that the machine functions as it was intended (to execute legacy binary-compatible programs) and to converge to commitment of program executions while following the Resource Flow execution model (presented in Chapter 3). We have also performed several simulation experiments on the OpTiFlow microarchitecture. These experiments can be divided into three main groupings.

Our first set of experiments compared the OpTiFlow microarchitecture to that of a conventional superscalar (the MIPS R10000 in the present case). We used the SimpleScalar MASE simulator for evaluating the conventional superscalar machine. Both machines simulated the execution of ten of the SpecINT-2000 benchmark programs. Both machines emulate the Alpha instruction set and executed the exact same benchmark programs compiled for that instruction set and to execute under the Compaq True-64 operating system run-time environment. Our simulator emulates enough of the Compaq True-64 operating system to achieve proper program executions on unmodified True-64 Alpha binary object files. All program were compiled by the Compaq native C language compiler and the exact same binary executable was used with the simulators used for each machine (both the conventional superscalar and OpTiFlow). The performance comparisons between the two machines (the baseline superscalar and OpTiFlow) were carried out on a variety of equivalently configured machines. Although each microarchitecture has a different hardware components and organizations and interconnections of its components, they both share some basic equivalent structures that allows for equitable performance comparisons. Both machines (in all configurations) shared identical memory hierarchy (main memory, L2 I&D caches, and L1 I&D caches) and branch predictors. Further, they both were configured with the analogous (or homologous) hardware machine component where they existed. Specifically, the both were configured for the same number of maximum instructions executing within the execution windows of each machine. They were also always configured with the same issue width and numbers and types of function units. Additionally, they shared parameters such as fetch width, decode width, and dispatch width for

all configurations. They also shared the same pipeline latencies for all function units configured. The OpTiFlow microarchitecture was also (additionally) burdened with a fixed number of operand transfer buses to forward operands from instruction to both the architected register file and other instructions. There is no analogous structure with the conventional machine simulation model. Also, the OpTiFlow microarchitecture suffered clock delays for arbitration of shared resources and a one clock delay for traversing operand transfer buses and the buses leading to and returning from the functions units. These bus delays are not present in the conventional machine model.

Our performance comparison results show that the OpTiFlow machine achieves approximately an IPC speedup of at least 2.0 over the conventional superscalar machine for all machine configurations explored. Harmonic mean (over all benchmarks) IPC speedups of the OpTiFlow machine over the baseline conventional machine range from 1.99 to 2.57 over a range of machine configurations having issue stations (or reservations stations) of from 32 to 128 and issue widths from 2 to 8. For most benchmark programs, the OpTiFlow machine somewhat proportionally followed the IPC performance of the the same benchmark program on the baseline conventional machine.

Our second set of experiments explored the numbers of instruction re-executions that the OpTiFlow microarchitecture uses in order to converge to program commitment for a variety of machine configurations. These results indicated that there is a rough or loose correlation between the IPC performance of a benchmark program and the number of instruction re-executions used to execute the program. More importantly, we observed a strong correlation between the number of instruction re-executions and the number of issue stations used within a particular machine configurations. Instruction re-executions did not well correlate with the ratio of issue stations to instruction issue width, as we might have first expected. These results and others indicate that large numbers of issue stations within the OpTiFlow machine (all of which share a single set of operand transfer buses) gives rise to many false operands (operands with values that do not eventually become committed values of the program) to cause unnecessary executions and re-executions for those instructions further away from the present instruction, in program-ordered time. This is a sort of destruction interference of operands that are being acquired by instructions as being feasible but incorrect (the operands come from the wrong preceding source instructions). This result illustrates that in addition to not being physically scalable (due primarily to the common central operand transfer buses), the OpTiFlow machine also has limitations for performance scalability even if physical limitations were overcome but where the same basic OpTiFlow microarchitecture remained (with the single common operand transfer buses). However, both physical scalability and performance scalability can be overcome and mitigated through an enhancement of the microarchitecture. This endeavor is taken up in Chapter 8, 9, and and 10 of this research work.

The third group of experiments performed evaluated two different strategies or policies for managing instruction re-executions within the issue stations of OpTiFlow (or any Resource Flow

based machine using issue stations). One instruction re-execution policy allows for completely unrestrained multiple overlapping simultaneous instruction executions for the same instance of an instruction within an issue station. The second instruction re-execution policy restricts the number of simultaneous instructions that can be executing at any one time for one instance of an instruction within an issue station to just one. Although the second policy might be assumed to generate significantly fewer (or substantially fewer) re-executions than the first policy – and perhaps achieving a higher execution performance as a result, we did not find this to be the case. Both policies generate almost the same number of instruction re-executions and also yielding approximately the same program execution performance. The second policy only very slightly achieved an average higher IPC execution performance over the first, but this not substantial enough to make a clear design decision for it in any machine implementation.

Table 6.5: *Harmonic mean IPC speedup of the OpTiFlow machine over the baseline conventional machine.* The harmonic mean IPC speedups of the OpTiFlow machine over the equivalent baseline conventional machine is provided. The harmonic mean IPCs for each machine configuration are computed over all benchmark programs. The

configuration	speedup
32-2	2.57
32-4	2.32
64-2	2.45
64-4	2.20
128-2	2.22
128-4	2.00
128-8	1.99
A-MEAN	2.25

Table 6.6: *Harmonic mean IPC speedups of the serial re-execution policy over the overlapping re-execution policy in OpTiFlow.* The harmonic mean IPC speedups of serial re-execution policy over the overlapping re-execution policy is given for a variety of machine configurations.

configuration	speedup
32-2	0.43%
32-4	0.94%
64-2	0.89%
64-4	0.87%
128-2	1.47%
128-4	0.48%
128-8	0.96%

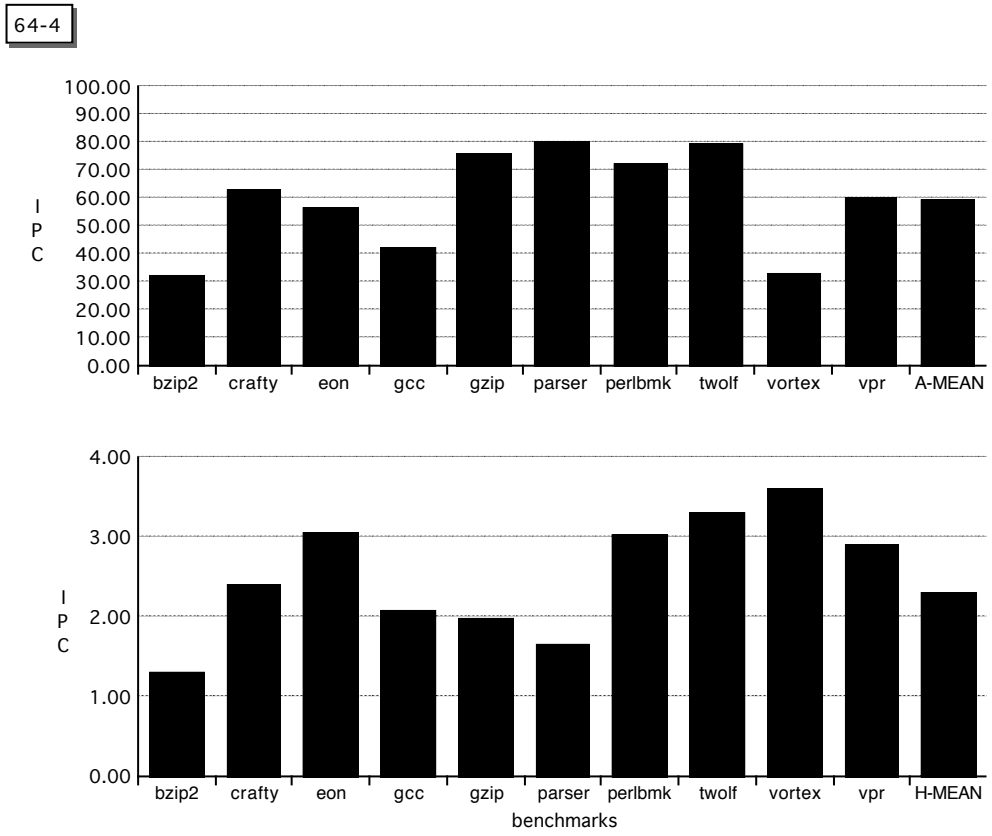

Figure 6.12: *The juxtaposition of instruction re-executions with IPC.* The top graph shows the percent instruction re-executions for the 64-4 OpTiFlow machine configuration. This is juxtapositioned with the graph of the same machine configuration below it.

Table 6.7: *Arithmetic means for percentage instruction re-executions for two re-execution policies.* The arithmetic means over all benchmarks for the percentage instruction re-executions are provided for each of the two re-execution policies examined. These results are shown for each of seven machine configurations. Machine configurations are given column one using the 2-tuple representation.

configuration	overlap	serial
32-2	39.6%	39.0%
32-4	40.7%	39.7%
64-2	58.8%	56.7%
64-4	59.3%	58.6%
128-2	81.7%	80.6%
128-4	83.4%	82.7%
128-8	86.2%	84.5%

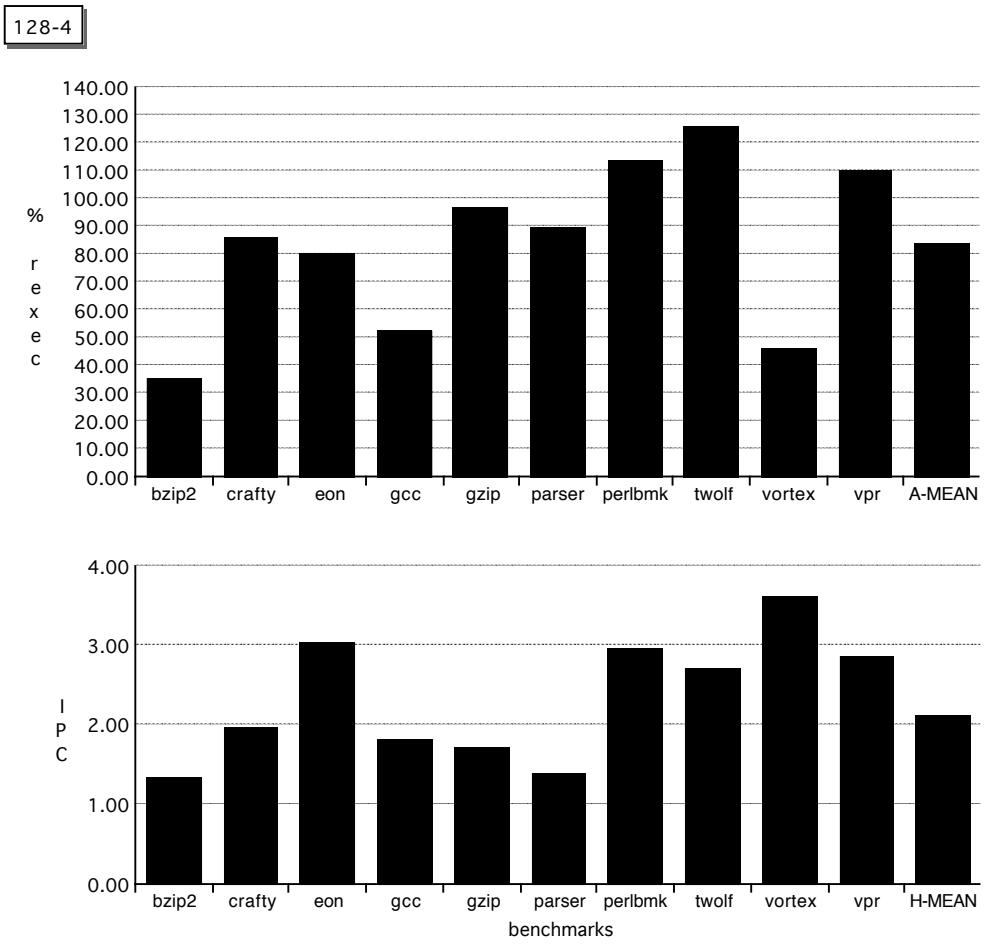

Figure 6.13: *The juxtaposition of instruction re-executions with IPC.* The top graph shows the percent instruction re-executions for the 128-4 OpTiFlow machine configuration. This is juxtapositioned with the graph of the same machine configuration below it.

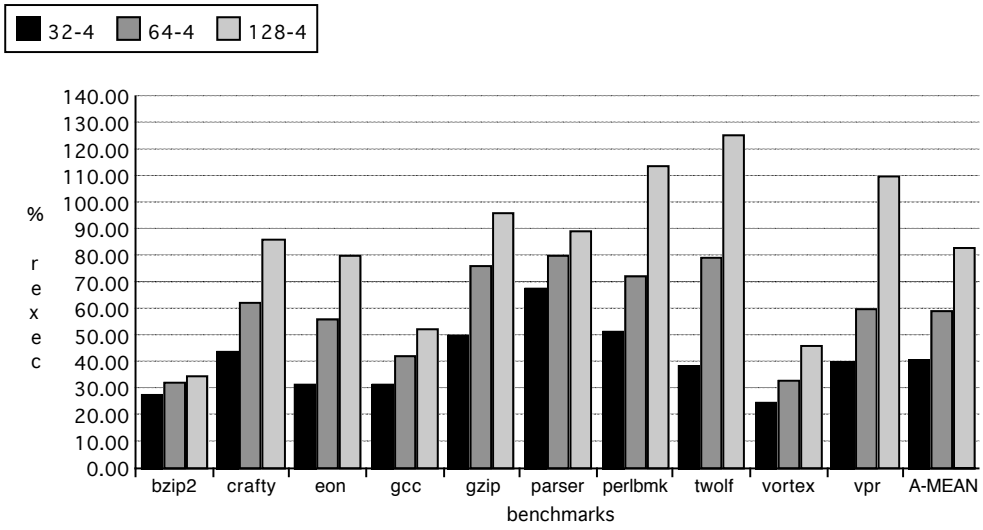

Figure 6.14: *The percentage instruction re-executions for three OpTiFlow machine configurations.* We show the percentage of instruction re-executions for three machine configurations. Machine configurations are indicated using the 2-tuple representation.

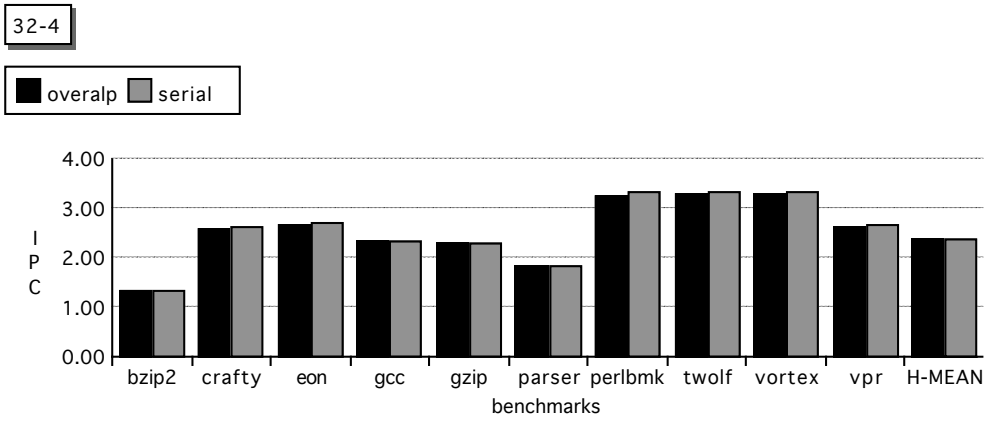

Figure 6.15: *Performance comparison of overlapping and serial re-execution policies in OpTiFlow.* An IPC performance comparison is shown for the overlapping and serial re-execution policies for the 32-4 machine configuration of an OpTiFlow machine.

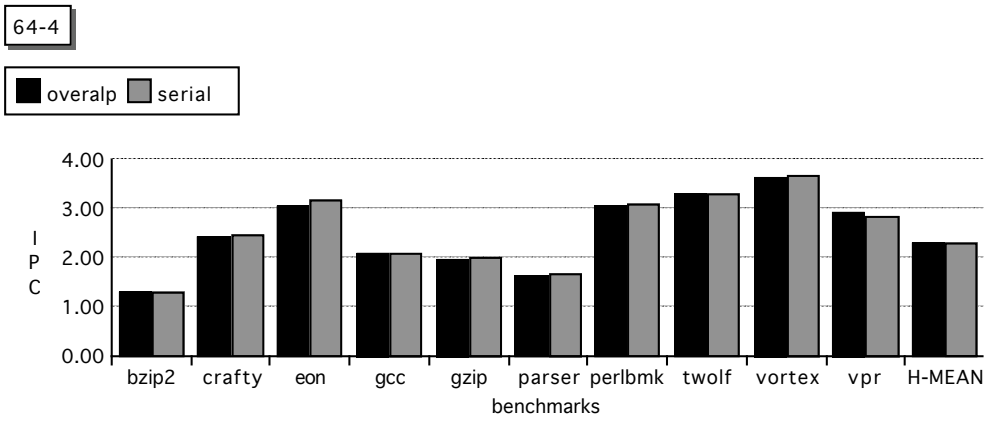

Figure 6.16: *Performance comparison of overlapping and serial re-execution policies in OpTiFlow.* An IPC performance comparison is shown for the overlapping and serial re-execution policies for the 64-4 machine configuration of an OpTiFlow machine.

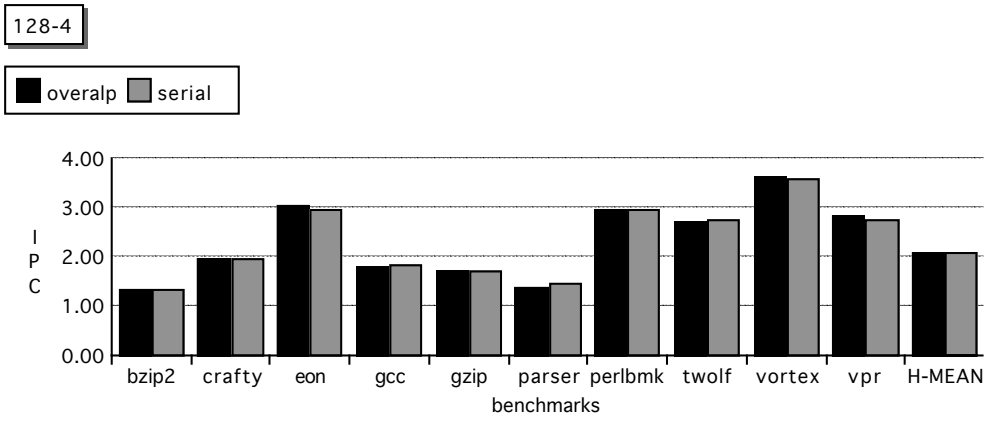

Figure 6.17: *Performance comparison of overlapping and serial re-execution policies in OpTiFlow.* An IPC performance comparison is shown for the overlapping and serial re-execution policies for the 128-4 machine configuration of an OpTiFlow machine.

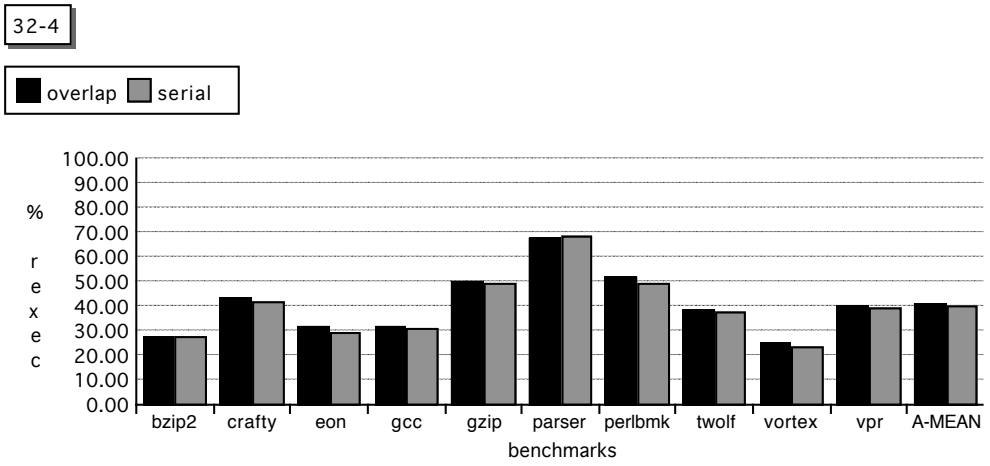

Figure 6.18: *The percentage of instruction re-executions is shown for the two re-execution policies examined.* The percentage of instruction re-executions for the `32-4` machine configuration is shown for the two re-execution policies examined.

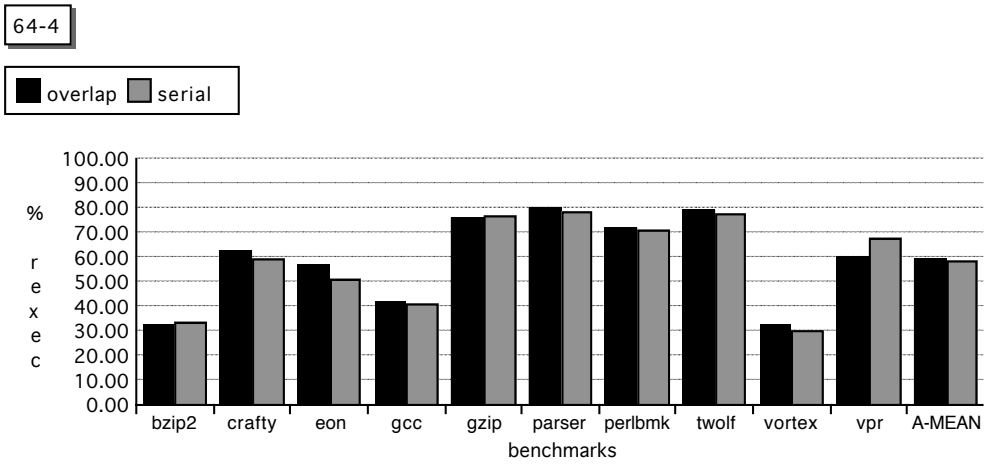

Figure 6.19: *The percentage of instruction re-executions is shown for the two re-execution policies examined.* The percentage of instruction re-executions for the `64-4` machine configuration is shown for the two re-execution policies examined.

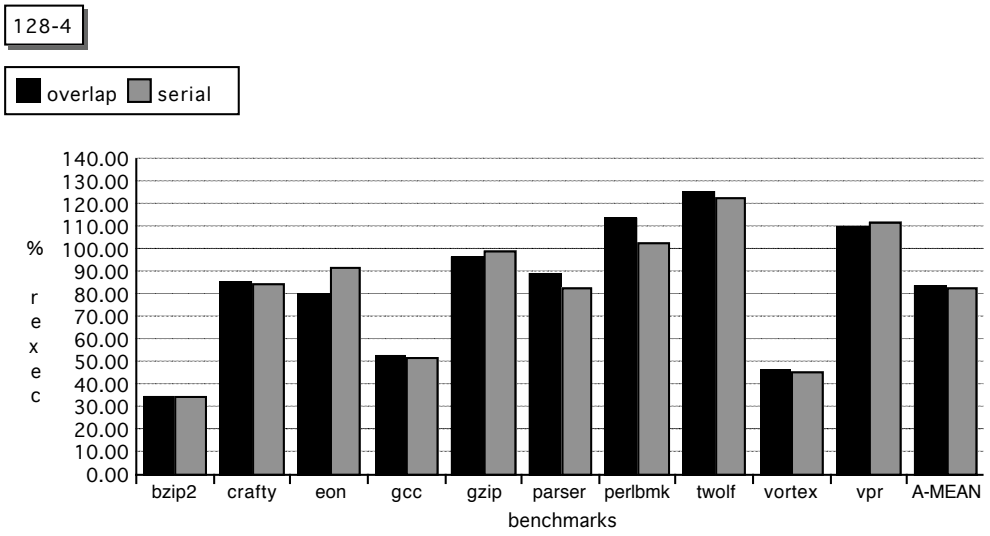

Figure 6.20: *The percentage of instruction re-executions is shown for the two re-execution policies examined.* The percentage of instruction re-executions for the `128-4` machine configuration is shown for the two re-execution policies examined.

Chapter 7

A new approach to microarchitectural predication

In this chapter we present a new variation on the concept of instruction predication. We introduce a new predication scheme that differs from those previously explored, presented in the research literature, or that have been employed in actual microarchitectures. Normally, predication is considered an architectural idea; namely, one that is visible to the programmer or compiler writer and is supported by the instruction set architecture (ISA) of the machine. Our present contribution is not one of an architectural nature but rather of a microarchitectural nature. However, our contribution to predication is able to support ISAs (an architectural feature of the machine) that may or may not have explicit predication support already. Our predication is not visible to either a programmer or a compiler writer and is therefore not of an architectural nature. Although our present contribution is not the first to introduce predication as purely a microarchitectural feature, our contribution is an improvement on existing microarchitectural predication techniques.

One of the main benefits of using predication (in any form) is its promise of execution performance gains. Microarchitectural predication (described more in subsequent sections) allows for both its adaptation to legacy instruction set architectures as well as possible increased performance gains by having predication dynamically applied to all executing code rather than just to code for which a programmer or compiler could have predicated. Finally, our enhanced microarchitectural predication scheme allows for better physical scalability of a resource flow oriented machine microarchitecture. The idea of physical machine scalability is discussed much more in Chapter 10.

The rest of this chapter is organized as follows. We first present a general introduction to the idea of predication. This includes an overview of the idea of predication and how it is used to govern or control the execution of a program. In addition, we introduce a taxonomy of various types of instruction predication. This taxonomy is useful to more clearly show how our contribution to

predication fits in with the existing ideas and contributions to predication. The next major section presents some background material that was the inspiration for our new scheme. The following section presents a general overview of our new predication scheme. This is in turn followed by a section that outlines the new state that must be added to the issue station component (described in some detail in Chapter 3) in order to implement our new predication scheme. A subsequent section provides some details about how the predication scheme works in the context of the Resource Flow execution model. We then have a section that provides some ideas about replacement policies for a predication structure (the branch target predicate table) that resides within the issue stations. We summarize in the last section.

7.1 Predication overview

Here we present an introductory view of instruction predication. Briefly, the idea of instruction predication is that an additional variable is associated with each statement in the program. That additional variable is termed the *predicate*. The predicate in this context is a binary valued variable that takes on the values of either TRUE or FALSE. The results of the predicated program statement only take effect (are committed to architectural state) when the value of the predicate is also determined to be committed as TRUE. Note that only the effects of the program statement are predicated on the predicate variable. Whether the statement speculatively executes or not is not of particular concern to the programmer. It may speculatively execute and hold its result tentatively until commitment or it may choose not to execute at all until the committed value of the predicate is determined. The case in which the instruction is allowed to speculatively execute but not commit (assuming that it was not predicated to commit) is sometimes referred to as *predicate slip*. [165] Other variations of speculative execution of the predicated instruction statement may also be possible but the primary idea is that the results of the speculative execution of a predicated instruction does not become a part of the program's committed state (and therefore visible to the programmer) unless the value of the predicate is determined to commit with a value of TRUE. Note that if the predicate variable itself may be determined speculatively, it is only its final value (the committed value) that is used as the determining predicate for its corresponding statement.

In brief, one of the goals for using predication is to try to eliminate conditional branch statements. This is generally desirable due to the difficulty of executing conditional branch instructions in existing machines. By eliminating some conditional branch instructions, a speedup of the program is usually possible since the associated microarchitectural processing of those conditional branch instructions is also eliminated. Conditional branches that occur in loops are obviously among the best obtained candidates for instruction predication, and likely lead to the most performance gains that may be had with predication. [22, 167] Note that in general it is not possible to

eliminate all conditional branch instructions using predication.

To illustrate the basic idea of predication, we present a program fragment in Figure 7.1. This is a program fragment that would be typical in ISAs that do not feature any instruction predication within the ISA itself. For our present purposes, instruction predication that is inherent within an ISA will be termed *architectural predication*. The example is purposefully not in any particular programming language in order to better facilitate transformation to a representative assembly language later. Expressions in the program fragment, as found in line 101 of Figure 7.1, are

line	statement
100	`a <= c1`
101	`if (b) goto 103`
102	`a <= c2`
103	`print a`

Figure 7.1: *Program fragment without using instruction predication within the ISA itself (architectural predication).* This is a familiar and typical program fragment that does not use any instruction predication through means of the ISA itself.

evaluated as in the C language. Lines 100 and 102 represent assignments to the program variable **a**. Values C1 and C2 (on right-hand-sides) can be considered constants. Note that for our present purposes, a program variable can be either a machine architected register or a memory location. Line 103 simply represents a use of variable **a**. The salient element of this program fragment is the conditional branch statement in line 101. That statement represents (or is translated into) a conditional branch instruction in the target ISA of the machine. It is this statement (the conditional statement) in particular that is eliminated through the process of predication. That is the goal of predication.

In terms familiar to high-level languages, the variable (or expression) represented as **b** in statement 101 is called the predicate of that statement. This variable **b** will be related to the idea of an architected predicate. This will be seen when this program fragment is transformed into an equivalent form using a form of architected predication. For convenience to the reader, the program fragment of Figure 7.1 is equivalent to the single C language statement:

print (b) ? c1 : c2 ;

Both the code form presented in the figure as well as the C language form may be useful for understanding subsequent transformations of this code fragment.

Figure 7.2 represents an equivalent form of the program fragment of Figure 7.1 showing the idea of predication that we are introducing. In this program fragment, we have introduced the predicate itself (the predicate variable) with the name **p1**. This represents one of several possible predicate variables that may be intermixed in any single fragment of code. For our present purposes, we

line	statement
100	`p1 <= (b != 0)`
101	`a <= c2`
102	`p1: a <= c1`
103	`print a`

Figure 7.2: *Program fragment using architectural predication.* This is an equivalent translation of the previous program fragment that did not use architectural predication.

assume that the predicate variable is an architected register, but this need not be strictly the case in any given ISA. However, in most RISC-like ISAs, the predicate is generally an architected register. In this example, there is only this single predicate variable. Statement 100 sets the value of the predicate (variable **p1**) with the logical result of the evaluation of the expression on the right-hand-side of the assignment statement. Note that we have taken the logical negation of the original variable **b** in order to set our introduced predicate variable. Carefully note statements 101 and 102. These statements are arranged to set variable **a** to the correct value depending on the value of variable **b**. Statement 101 is executed unconditionally, but statement 102 is only executed (or apparently executed so as to affect the resulting program architectural state) if the value of the predicate **p1** is TRUE. This arrangement results in the equivalent result as obtained with the program fragment of Figure 7.1 but without the need for a conditional branch instruction.

The program fragment shown in Figure 7.2 is not the only possible predicated version of that shown in Figure 7.1. Many other variations are also (perhaps obviously) possible. An additional equivalent predicated program fragment is shown in Figure 7.3. The program fragment of Figure

line	statement
100	`p1 <= (b == 0)`
101	`a <= c1`
102	`p1: a <= c2`
103	`print a`

Figure 7.3: *Additional equivalent program fragment using architectural predication.* This is an additional equivalent translation of the program fragment shown in Figure 7.1. Note the difference between this version and that shown in Figure 7.2.

7.3 illustrates the flexibility of choices for which statements can be predicated to achieve desired results.

Note that in both the predicated program fragments of Figures 7.2 and 7.3 the newly introduced predicate variable (**p1**) was always used in the positive sense where it predicated an assignment statement. This need not always be the case, but it is an architectural feature of the ISA designed to employ this type of predication. An example of an equivalent program fragment for an ISA that

allows for predicated statements using a negated predicate is shown in Figure 7.4. In general, some

line	statement
100	`p1 <= (b == 0)`
101	`a <= c2`
102	`! p1: a <= c1`
103	`print a`

Figure 7.4: *Additional equivalent program fragment using the negation of a predicate variable.* This is an additional equivalent translation of the program fragment shown in Figure 7.1. This version uses the negation of a predicate variable in the predication of a statement. Note the difference between this version and that shown in Figure 7.3.

ISAs may allow for both the positive and negative (negated) use of predicate variables through the allocation of an additional bit in the instruction encoding in order to indicate the appropriate predicate sense. For our purposes, we only employ the idea of positive uses of predicate variables with no loss of generality.

Although an ISA might be designed to allow predicate variables to be memory locations, this has not been done nor is it likely to be done. Using memory locations as predicates at the ISA level of abstraction would be very costly in terms of instruction encoding space (barring some prefix-instruction scheme such as employed in the x86 ISA for other purposes), so it is likely that only registers will ever be employed as architected predicate variables. Note that the number of predicate variables is still an open design issue, but because predicates need only occupy a single bit of architected register space, a sizable number of predicates can easily be employed in the modest of ISAs. Having even 64 predicate registers is not unreasonable, especially if the rest of the ISA is 64-bit oriented (such as iA-64).

7.1.1 Predication taxonomy

The idea of predication discussed in the last section represents only one of several possible implementations of predication in computing. In general, the idea of instruction predication may enter into use through a variety of means or at different levels in the whole of the machine execution model. It is not easy to discuss predication in general since the idea is not well developed in microarchitectural research (or practice) to date. For this reason, we first present a brief taxonomy of the idea of program predication before explaining what our contribution is centered around. Since terms for various ideas of program predication are not generally established, we introduce some terms ourselves and describe what they refer to as we proceed. Taxonomies other than that presented here are certainly possible; however, we need something at hand so as to provide some context for the following discussion. For our purposes, we first classify predication into two main

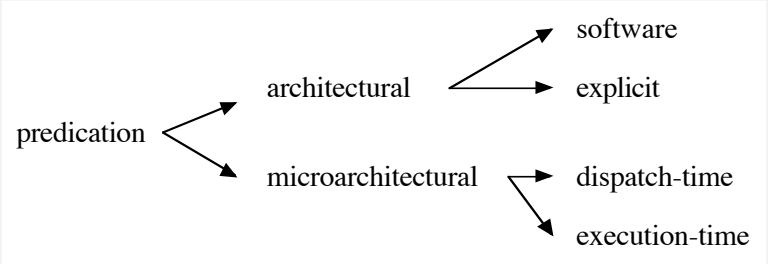

Figure 7.5: *A simple predication taxonomy.* A primary distinction is made in types of predication with a further subdivision of each of the main branch types.

categories. These are *architectural* and *microarchitectural*. A schematic depiction of the various predication types (terms) is presented in Figure 7.5.

Our idea of architectural predication is that the programmer is aware of it and has arranged for it explicitly through the choice of instructions within the given instruction set architecture that she has to work with. For our purposes, the programmer can be either an assembly or machine language programmer, or a compiler writer who is concerned with the choice and scheduling of appropriate instructions from the given ISA to implement the higher language program design intent. Predication of this type is of an architectural nature since it is realized through the use of the architecture of the machine, and more specifically through both the use of the ISA of the machine and some architecturally visible (visible to the programmer) machine registers. The general goal of architectural predication is to facilitate the elimination of conditional branches from the instruction flow. This usually results in increased execution performance of the program since the problems associated with a control-flow split in the instruction stream following the branch are eliminated. Normally, it is not possible to eliminate all conditional branches through predication (for example: consider looping control-flow constructs) but the more conditional branches that can be eliminated the better the chances are for some execution performance improvement. The principal problem associated with conditional branches for the underlying machine is that of managing the two possible future instruction streams emanating from the branch. Machine design choices about how to best handle the two speculative instruction flows from the branch can be complicated and performance limiting. However, these machine design and performance problems are made moot by eliminating the conditional branch instruction itself through predication. This general type of predication can also be referred to as *static predication* (in like manner as static branch predication is defined). Architectural predication is further broken down into *architectural software predication* and *architectural explicit predication*. The distinction between these may be considered a fine one, but both have been used at various times and the latter has come into more recent widespread use due to the introduction of a processor featuring it.

Microarchitectural (or non-architectural) predication is so named because it is entirely implemented within the microarchitecture of a machine. Specifically, there are no user-mode visible registers or instructions that are used for the implementation of this type of predication. All microarchitectural predication types (of which there can be many) can therefore be implemented with any machine ISA. This is possible since it does not require any changes to an existing ISA in order to be implemented. Further, one implementation of an ISA may not have any microarchitectural predication (perhaps for design simplicity reasons, power savings, or other economic reasons) while another may implement it. In either case, the exact same binary-compiled programs are expected to execute on the respective machines, only with possible performance differences occurring. This general type of predication can also be referred to as *dynamic predication*. This name (having the word *dynamic* in it) is somewhat analogous to the idea of dynamic branch predication as opposed to static branch prediction. Continuing with this analogy, static branch prediction (being implemented by the use of the ISA) would correspond with those types of instruction predication that would use features of the ISA. While dynamic predication would not use any features of the ISA, and rather be implemented entirely within the microarchitecture. For our present purposes, microarchitectural predication is subdivided into *microarchitectural dispatch-time predication* and *microarchitectural execution-time predication*. Although both are implemented within the microarchitecture of a machine, they differ as to the stage of the machine in which the discovery of the necessary predicate relationships (or control flow dependencies) among the in-flight instructions occurs. As the names somewhat imply, dispatch-time predication determines the necessary predicate relationships at the time that instructions are dispatched (after being fetched and decoded), while the execution-time predication technique discovers the necessary predicate relationships during the course of execution (or more specifically after instruction issue). More information on dispatch-time predication is provided below in both the brief taxonomical description of that predication type as well as in the background section.

Each of these four predication types is briefly discussed in the sections below, and provides some orientation for understanding our contribution. Our contribution presented in this chapter is a form of microarchitectural execution-time predication but which resembles, inside the microarchitecture, the flavor of the structure of the architectural explicit predication type.

Architectural software predication

Here we discuss the idea of instruction predication that is implemented by means of the ISA of the machine, but which is not done so in a way where each instruction is explicitly predicated with its own predication variable. We term this type of instruction predication *software predication*. Software predication in the context of non-dataflow machines exploits certain traits of an

ISA to effectively perform conditional assignments but without having to use conditional branch instructions. A related idea for data-flow machines is discussed later in this chapter where a sort of special case of predication is employed. Software predication should not be confused with the expression that is evaluated in a conditional statement of a high-level language (often termed the predicate of the conditional). Rather, this is a type of predication that employs the capabilities of the architectural machines' ISA in creative ways. Consider the goal of creating a subroutine that takes the absolute value of its single integer argument and returns that value from the subroutine. An example of such a subroutine as written in a high-level language is shown in Figure 7.6. The

line	statement
100	`int abs(int v)`
101	`if (v >= 0) goto 103`
102	`v = (- v)`
103	`return v`

Figure 7.6: *Subroutine to return the absolute value of its integer argument.* This subroutine is a typical high-level language implementation of a simple subroutine that returns the absolute value of its single integer argument.

subroutine program of Figure 7.6 might be translated into assembly language as shown in Figure 7.7. The assembly language shown is actually for the M680X0 family of processors and is an actual working example for that ISA. The assembly language code of Figure 7.7 is straightforward and

line	statement
abs	`movl 4(%sp),%d0`
101	`tstl %d0`
102	`bge 104`
103	`negl %d0`
104	`rts`

Figure 7.7: *An assembly language subroutine for taking absolute value.* This is an actual assembly language subroutine using the M680X0 ISA. It returns the absolute value of its single integer argument as passed on the stack from a C language caller. The result is returned in register D0 according to the employed subroutine calling convention.

typical of such functions when not using software predication. No subroutine frame establishment or register saving and restoring is needed for this particular subroutine since neither automatic memory variables nor additional registers beyond those that do not require saving are used in its implementation. However, the use of the conditional branch in statement 102 is constraining to the fast execution of this subroutine. This subroutine is both small and used often in certain compute-intensive applications. For this reason, some additional investment in making this subroutine (for example) faster may be warranted. A functionally equivalent subroutine, also in M680X0

assembly language, is shown in Figure 7.8. The subroutine program shown Figure 7.8 uses software

line	statement
abs	`movl 4(%sp),%d0`
101	`movl %d0,%d1`
102	`addl %d1,%d1`
103	`subxl %d1,%d1`
104	`eorl %d1,%d0`
105	`subl %d1,%d0`
106	`rts`

Figure 7.8: *A subroutine showing the use of Software Predication.* This is an actual assembly language subroutine using the M680X0 ISA and which employs Software Predication to avoid the use of conditional branch instructions.

predication to avoid the use of the conditional branch instruction that was used in statement 102 of Figure 7.7. This subroutine also does not require frame establishment or any register saving and restoring since both registers D0 and D1 (the only two registers destroyed) do not require saving in the given subroutine-calling convention. At first, it may not be apparent how the use of a conditional branch instruction was avoided but careful examination reveals how this subroutine works. The first statement copies the passed argument into register D0 (the exact implementation of this is something that is dependent on the subroutine calling convention in use). Statement 101 creates a copy of this into register D1. Statement 102 effectively sets the architected carry-out bit to the sign bit of the passed argument. Statement 103 creates a mask of either all zero bits or all one bits in register D1 depending on the state of the carry-out bit previously determined. Statement 104 uses the created bit mask to conditionally take the ones complement of the passed argument (currently in register D0), depending on whether the passed argument was positive or negative. It does this through using an exclusive OR instruction using the created mask in register D1 against the working value in register D0. Statement 105 conditionally (according to the value in register D1) modifies the running value in register D0 to a twos complement from its previous ones complement. The resulting value in register D0 is returned. If the original value was negative, the result is to return its twos complement (its negative). If the original value was zero or positive, it is returned as it was. Although this subroutine used more instructions (two additional instructions) as compared with the program shown in Figure 7.7, it actually executes faster than that previous version. The amount of the speedup is dependent on which particular processor within the MC680X0 ISA family of processors is used. Generally, different processors, even within the same ISA family, execute instructions and instruction combinations faster or slower than other processors depending on their internal microarchitecture. Fortunately, additional details about particular processor microarchitectural implementations are not important to our present discussion. The

primary reason for the speedup is the elimination of the conditional branch instruction that was used in the previous version. This latter subroutine was a somewhat more complicated example of the use of software predication. In general, examples of this complexity are not easily created. But this example does show the potential of using software predication for eliminating conditional branches and achieving a resulting speedup in the process, despite possibly requiring the addition of more instructions (as were needed in this example).

A more typical example of software predication (and more modern) is represented by the program fragment shown in Figure 7.9. In this example, pseudo (not an actual) assembly language is

line	statement
100	`r1 <= (b == 0)`
101	`r2 <= c2`
102	`r3 <= c1`
103	`! r1: r2 <= r3`

Figure 7.9: *A more typical example of software predication using the Alpha ISA.* This is an additional equivalent translation of the program fragment shown in Figure 7.1. Pseudo assembly language is used for additional clarity. This version uses the negation of a predicate variable in the predication of a statement.

shown for clarity. More specifically, registers R1, R2, and R3 are regular general purpose registers in the Alpha architecture and statement 100 simply evaluates an expression and places the result in the general purpose integer register R1. the register R1, although a general purpose integer register, is effectively used as the predicate. Statement 102 is actually a single instruction (the CMOVE instruction) in the Alpha ISA. [68, 67] The Alpha CMOVE instruction is short for *conditional move* and gives a hint to the conditional nature of its semantics. The idea of a conditional move instruction was also a feature of other proposed machine architectures such as the Very Long Instruction Word (VLIW) architecture by Rau [105].

Architectural explicit predication

This is an architectural predication type like the software predication type discussed above, but with this type explicit predication registers are provided within the machine architecture. These registers can be either existing general purpose registers (most likely the integer registers) or they may be additional (in addition to the general integer and floating point registers) special purpose registers used just for carrying out the predication functions within the ISA. When special purpose predication registers are provided they are usually only single bit registers, since only a single bit is needed to retain the result of a high-level language predicate evaluation. In this predication type, each architected instruction now also has associated with it a reference to that predicate register that should guard the present instruction. This reference is a small field of bits that encodes the

architected address of the desired predicate register into the overall encoding of the instruction. Of course, the encoded field will have a number of bits corresponding to the power-base-two ceiling of the log of the number of architected predicate registers.

An example of a machine architecture using this form of predication is the iA-64 machine architecture [119]. That machine architecture features 64 single-bit special predicate registers that are then specified in its instructions either as the recipient of a predicate evaluation or as the predicate to be evaluated later in assignment or other instructions meant only to have conditional committed results. Another machine to include a predicate specification for each of its instructions is the Cydra 5 machine. [106]

Examples of pseudo assembly language for this predication technique can be the code fragments of Figures 7.2, 7.3, and 7.4. The semantics of this predication type follows the general discussion of the predication concept we have already introduced. Obviously, this type of predication closely follows the conceptual idea and was purposefully intended to do just that. Remember that this predication is an architectural one (one that is architecturally visible in the machine) and might be used by either the code scheduler in a compiler or directly by an assembly language programmer. Therefore the basic predication ideas purposefully flow very directly into the semantics of the ISA of the machine.

As alluded to previously, we also note that some ISAs can support both the use of positive logic predicates as well as negative logic predicates. This capability might be useful for the efficient representation of larger and more complex cases of double-sided Hammock conditional branch schedules, translated from (for example) the if-then-else constructs of a higher-level language.

Besides associating a reference to a predicate register with each instruction, as we have assumed up to this point, sub-parts or sub-operations of a single instruction might also be predicated explicitly. This may seem obscure but it is not infeasible when complex instructions that manipulate many architected output registers (or memory locations) are considered. An example of this idea being implemented in one form was with the Cray-1 supercomputer. [112] That machine implemented instructions (called *vector instructions*) that operated on a whole array of operands with just a single instruction (a complex instruction). It would often be desirable to exclude one or more of the operands with the specified array from being affected by the operand of the instruction. This machine allowed for selective update (or non-update) of certain operands with the operand array through the specification of bits in a bit array. This bit array essentially formed an array of architected predicate bits. However, instead of determining whether a whole instruction was to be committed or not, individual operands of a vector instruction could be selected for commitment. The conditional select mechanism within the Multiflow machine architecture [88] is not dissimilar. It allows for parts of composite instruction (in the context of a VLIW architecture) to be predicated.

Before leaving architectural predication we note that architectural predicates are simply single-bit architected registers. These can be thought of as additional architected registers within an ISA but which derive their value from the result of a conditional branch instruction execution of some kind (whether that conditional branch instruction is executed speculatively or not). Just as outcomes of conditional branch instructions can be predicted, so can the associated single bit predicate register reflecting the outcome of its branch. Prediction of architected predicate registers was described by Chuag et. al. [16]

We have seen that the present predication type (architecturally explicit predication) expresses a degree of explicitness since it associates a predicate variable with each instruction that is predicated. This same sort of explicit association of a predicate variable with a corresponding predicated instruction will also be present in the microarchitectural predication types that will be discussed next.

Microarchitectural dispatch-time predication

As its name suggests, this predication type occurs both within the microarchitecture of the machine (not architecturally visible) and also determines the predicate relationships among the instruction flow at dispatch time. Presumably, other microarchitectural predication types can occur at times or stages within the machine other than at dispatch time. Indeed, our new predication contribution is one such predication type. But for our present purposes, we now further differentiate two subtypes of dispatch-time predication.

The first of these is the familiar speculative dispatching of instructions beyond a conditional branch instruction in many conventional machines that employ either register update units (RUUs) or reorder buffers (ROBs). In both of these subtypes, an implicit control-flow relationship is established between the conditional branch instruction and those subsequently speculatively dispatched instructions following it in program order. The relationship is recorded within either the RUU or the ROB by virtue of the spatial relationship between the placement of the tokens representing the respective instructions. Specifically, speculatively dispatched instructions following the conditional branch instruction are placed in either the RUU or the ROB after the conditional branch instruction itself. Note that the order of instructions placed within either an RUU or ROB are always maintained in a FIFO relationship, in accordance with the program order of the original instructions. This allows for the necessary control-flow relationships to be retained and used in the event that the conditional branch instruction resolves such that the subsequently dispatched instructions need to be squashed. Since the control-flow relationship of instructions placed in a RUU or ROB is determined by the FIFO management of those machine structures and not due to any explicit dedicated predicate register, this type of predication can be termed *microarchitectural*

dispatch-time implicit predication. Although this type of instruction predication has long been employed in machines, good reasons to name it as we have here have not generally been necessary since there were not very many alternatives available for which good differentiation was required – until now.

A second and more interesting (and perhaps more important) type of dispatch-time predication is that which employs a more explicit means to represent the control flow relationships among the instructions in-flight within the machine. The more explicit means is to use predication registers in a manner that resembles what was used in the architectural explicit predication described previously. Except now the predication registers are not architecturally visible but rather are maintained completely within the microarchitecture of the machine. As with the architecturally explicit predication, a one-bit predication register is maintained as state, along with the instructions that are dispatched. This predication register is simply a part of other state that may be maintained with the dispatched instruction during its flight within the machine. Other state can include things like decoded operation codes and tags that point to speculative register entries (in either an RUU or ROB for example) for both input and output registers. In this predication subtype, control flow dependencies (predicate relationships) among dispatched instructions are determined at the time of the dispatch through the means of a coordinating predication table. Although whether an instruction's predicate is enabled (allowing for normal commitment) or disabled (preventing commitment) can be changed dynamically during the execution (and possible re-execution) of a given instruction, the relationship among the predicate registers associated with each dispatched instruction is only determined at dispatch time. It should be noted that this predication subtype is still exceedingly dynamic in its operation as compared with the static architectural predication types described previously. However, the ability to also determine the control-flow relationships among the instructions at issue or execution time represents an even greater degree of dynamics, which also happens to lend itself to a simpler hardware design implementation. Since this predication subtype employs an explicit (although microarchitectural) predication register that is associated with each dispatched instruction, it is termed *microarchitectural dispatch-time explicit* predication.

More details on this predication subtype is presented in the background section below. The first known predication scheme of this type was described by Uht [155]. Uht termed his predication scheme *hidden explicit* predication. His use of the term *hidden* was meant to denote its type as being microarchitectural. This predication subtype also serves as the primary inspiration for a non-dispatch-time predication type that is the subject of this chapter. That type is briefly described next.

Microarchitectural execution-time predication

For our present purposes, this predication type maintains most all of the attributes of the microarchitectural dispatch-time explicit type described above. It differs in that the control-flow relationships among dispatched instructions occur during execution time rather than at dispatch time. This predication type was directly inspired from the dispatch-time explicit predication type just described (and which was first introduced by Uht). A scheme to implement this type of instruction predication is the contribution of this chapter.

7.1.2 Orientation and overview of our contribution

In the subsequent sections we introduce a new predication scheme of the microarchitectural execution-time type. This new scheme builds on the work of a previous microarchitectural predication scheme (that of Uht) but improves upon it by relieving some of the hardware design problems associated with that scheme. More specifically, our predication scheme allows for the elimination of a centralized hardware machine structure altogether and also facilitates better physical machine scalability.

Our predication scheme also assumes the resource flow execution model that has already been presented in Chapter 3. The idea of the issue station (IS), first introduced with the idea of resource flow computing, plays a central part in our scheme. We show how additional state information attached to the issue station, along with the management of that state through additional issue station logic, is all that is necessary to implement our scheme. One item of the state associated with an issue station is the so-called instruction predicate itself – the one bit register that determines whether the effects of the instruction are to be committed or not. So far, this arrangement at a high level is not that much different than that which would be required by the Uht scheme. However, our new predication scheme differs from Uht's in that the addresses of the predicates (or issue stations) and the predicate values are entirely determined at execution time (after instruction dispatch) rather than before or during instruction dispatch as was done in Uht's scheme. Our scheme also entirely eliminates the need for any centralized hardware structures as is needed by Uht's scheme. This provides for a less complex hardware implementation than was previously possible.

The remainder of this chapter is organized as follows. In the following sections we first present some background information on the predication scheme that served as the inspiration for our present contribution. In that context, we discuss both the implementation problems with that scheme and the objectives of our new scheme to solve those problems. We then have a section that presents an overview of our new predication scheme. This is followed by a section that discusses the state that must be added to (or associated with) each issue station in order to implement our scheme. We then discuss the operation of the new scheme in a subsequent section. We also have a small section on possible replacement policies for a hardware structure of the new scheme. This is

followed by a short section on other management possibilities related to our dynamic predication scheme not already discussed. We close with a summary of the contribution.

7.2 Background

The idea of predication can be thought of an extension of the idea of control-flow dependency enforcement. Proceeding from this vantage point, the earliest machines that contained conditional branching instructions employed, in a sense, a rudimentary form of predication. More specifically, the first instructions of the two or more basic blocks constituting the outcomes of a condition branch instructions were in effect predicated within the microarchitecture. They were predicated on the outcome of the conditional predicate within the conditional branch instruction preceding them. This may be considered a degenerate form of predication, but with the advent of data-flow machines (architectures with associated microarchitectures) this form of predication (really control-flow dependency enforcement) had to became more explicit (in both the architecture and the microarchitecture). The reason data-flow machines needed to handle this more explicitly was due to the fact that they attempted to almost totally relax any requirements for sequential execution of their program instructions. In data-flow machines, the idea is to allow (or to try to allow for) all instructions in the target program to execute without regard for their static order within the compiled program. However, within this framework, both data flow dependencies and control flow dependencies (being understood as only being the true control flow dependencies and not those simply arising from static order of the compiled program instructions) must still be observed and enforced within the microarchitecture. For our present purposes, this idea of how data-flow machines enforce control-flow dependencies is a useful starting point for the development of predication within microarchitectures. Data flow machines convert control-flow dependencies into something similar to a data-flow dependency. It does so in the context of allowing massively relaxed (or unrestrained) order of instruction execution. This happens to be the same as our goal within the development of our Resource Flow execution mode.

The earliest schemes to convert control-flow dependencies into data-flow dependencies were first introduced in the context of data-flow machines. These are now termed *partial predication* schemes since only a single instruction, of a stream of dependent data-flow instructions, is predicated. The earliest work of this nature was done by Dennis et. al. [24] and Allen et. al. [2] A more recent example of this type includes that by Arvin et. al. [5] However, all of the work on predication within data-flow machines is more about simply enforcing the necessary (or true) control-flow dependencies in the program rather than really introducing the idea of predication as was discussed in the introduction to this chapter above. For our purposes, real instruction predication is the possible allowance of speculative execution of both portions of the program following a conditional branch

(the basic blocks following the possible dynamic output paths of a conditional branch) whether those instructions are destined to be part of the final committed program state or not (being abandoned). In this context, data-flow machines do not implement predication in the sense that we desire (allowance for possible arbitrary speculative execution of instructions) but rather, as already stated, they are really just implementing something to enforce control dependencies. As we have discussed in the introduction (and outlined with our newly introduced predication taxonomy), approaches that do implement predication in this sense may involve software, architectural, or microarchitectural means.

There have been approaches for allowing the equivalent of dynamic instruction predication at execution, but only for specific program instruction constructs. Two of these approaches were specifically targeted to try to deal with the instructions following a single-sided Hammock conditional branch (a conditional branch where both output paths join before encountering any additional unconditional or conditional branches). There approaches were proposed by Klauser et. al. [72] and more recently by Sankaranarayanan and Skadron [115] but were never implemented. A more aggressive approach towards implementing dynamic predication has been proposed by Kim et. al. [70] but this approach bears similarity to a partial predication scheme rather than a full or arbitrary one. We would like to note here that each of these three methods just mentioned to support a form of dynamic instruction predication have what can arguably be considered very complex and elaborate logic in the microarchitecture to support their approaches. Rather arbitrary, involved, and complex logic within the microarchitecture just to support some limited predication is also something that we would like to avoid with our work, and we feel that we have achieved this – to a much greater degree – than any of the approaches mentioned. Further, as already stated, an additional goal of our work over everything thus far is to provide a generalized microarchitectural scheme for arbitrarily predicating (at execution time) any or all instructions that are in-flight.

The first generalized microarchitectural predication scheme (our present interest) that does allow for possible speculative execution of arbitrary instructions (more than just minimal control flow dependency enforcement) was described by Uht et. al. [155] This scheme allows for all instructions in-flight within a machine to be predicated regardless of the number and variation of control flow changes within the instruction stream. By having all instructions predicated the scheme allows for the maximum amount of both out-of-order and speculative execution within the microarchitecture itself. In the scheme introduced by Uht, microarchitectural instruction predication is accomplished by calculating control-flow dependencies for predicates associated with each instruction and the initial values of those predicates at instruction dispatch time (when decoded instructions are transfered from the I-fetch buffers to instruction issue slots). This scheme requires the presence of a *Branch Tracking Buffer* that is used to correlate branch target instructions with the original branch instructions that lead to them. This hardware component has not been implemented in less than

$O(n^2)$ time or space due to the need to search the entire table simultaneously for all of the instructions to be dispatched (transfered to issue slots) in any given clock period. Our present goal is to eliminate both the need for Uht's central table and the circuit routing and searching complexity associated with it.

The new scheme that we introduce here is a natural extension of the one originally proposed by Uht [155]. In the scheme by Uht, the idea of an issue station was introduced as a mechanism to hold the related state associated with a dispatched instruction until it gets retired. An issue station (IS) is essentially the combination of an instruction issue slot and a reorder buffer entry of many current superscalar machines. This combined arrangement of hardware allows for the repeated executions of the dispatched instruction until it is determined that it should be retired, either committed or squashed as the case may be. The idea of the IS is a useful one for maintaining state associated with an instruction that has been dispatched. Each IS maintains a special bit of state (in addition to other state) termed an *execution predicate*. Again, an execution predicate is that which determines whether the current instruction's computed output values should become a part of the committed program state or not. If the instruction's execution predicate is TRUE, then the instruction is said to be *execution-enabled*. If the instruction's execution predicate is FALSE, then the instruction is said to be *execution-disabled*. If the instruction was execution-enabled at retirement, then it is committed. If the instruction was execution-disabled at retirement, then it squashed. Note that the term *execution predicate* is somewhat misleading in that an instruction is allowed to speculatively execute even if it is currently disabled. In this case, the hardware must hold any instruction execution results tentatively since they cannot become a part of the program committed state if the present instruction is retired while still in the disabled state. Of course, if the instruction becomes enabled, any current instruction execution output values may still be valid and may be used as long as no new inputs to the instruction have changed since the last execution. Whether instructions are allowed to execute while they are disabled is a machine implementation policy and is not of further interest to the work presented here. In this present work, we have adopted the notion of the issue station as a means to describe the state associated with an instruction that has been dispatched.

In contrast to the scheme by Uht, rather than computing any predicate dependencies (the predicate addresses) and their initial predicate values at instruction dispatch time (the time that instructions are dispatched to the ISes in the execution window), our new scheme would simply dispatch the instructions to issue slots without any calculated predicate dependency addresses. Control dependencies, in the form of execution predicates, are determined as instructions execute in a similar way as to how register and memory dependencies are determined dynamically at execution time. The idea for dynamically determining the predicate dependencies at instruction execution time, rather than at fetch or dispatch time, was first suggested by Alireza Khalafi. [63]

Khalafi observed that if register and memory dependencies could be determined dynamically at instruction execution time, and if control dependencies were converted into what appeared to be data dependencies within the microarchitecture of the machine (using microarchitectural predicates), then the dependencies on these predicates should also be able to be determined dynamically at instruction execution time. Why wasn't this indeed realized from the outset?

7.2.1 Problems with the use of a predicate tracking buffer

The most significant problem with the previous scheme is that a centralized predicate tracking buffer must be maintained. Although this is not a problem at all for small and more conventional microarchitectures, it is more of a problem with a large (issue window size) or distributed microarchitecture (a design space that we want to address). In large machines, and when using Uht's predication scheme, a large number of instructions generally need to be predicated at once in a single clock period. This has proven to be very difficult to do with the centralized predicate tracking buffer. Each new instruction needs to associatively search the predicate tracking buffer simultaneously within a single clock. This leads to order h^2 connectivity within the tracking buffer, where h is the number of instructions in the microarchitecture that need to be predicated simultaneously at dispatch time. A further complication with a centralized tracking buffer is that predication of later instructions in a column depend on the predication and buffer updates from earlier instructions in the column. These various problems and the hints acquired from the handling of register and memory dependencies has led to the development of an execution-time scheme to predicate instructions.

7.2.2 Objectives of the new scheme

The new predication scheme needs to not only avoid the use of a centralized microarchitectural structure (the predicate tracking buffer for example) but also must satisfy other requirements that are consistent with a flexible instruction dispatch policy. A scheme needs to be independent of the distance (in instructions) between a conditional branch and the target of that conditional branch. When instructions are dispatched into the execution window following the static program order, the distance between a conditional branch and its target equals the difference in their assigned IS time-tags. However, when instructions are dispatched into the execution window following the taken output path of a conditional branch, the distance in dispatched instructions from the branch to its target may be zero. Other dispatch policies (not elaborated on further here) could possibly result in some pseudo random distance (in instructions) between the branch instruction and its target. Therefore a predication scheme must be insensitive to the distance, in ISes, between a branch and its target. Unlike the previous scheme, no "null-predicate" or predicate-only issue stations need

be allocated and managed at instruction dispatch time (or otherwise) for the overflow of branch targets to a single instruction. Finally, the scheme must be insensitive to the real-time ordering of predication forwarding transfers. This is necessary as the forwarding bus network may allow forwarding transfers to slip past each other as they are forwarded.

7.3 Overview of the new scheme

In this section, we provide a brief outline of the scheme that we present. The context of a Resource Flow execution model is assumed. Specifically, the existence of time-tags for dependency ordering plays a crucial role in the working of the scheme. We first briefly discuss the predicates that are evaluated for determining whether an instruction is enabled or not. Next we present an overview of how our predicate scheme interacts with the order of instructions flowing in-flight through the microarchitecture.

7.3.1 What are the predicates?

One of the central goals of the present scheme is to determine predicates for instructions at execution time rather than at fetch or dispatch time. This is why nothing about the assignment of predicate registers has been discussed so far above. Rather, after instructions are dispatched to ISes and are therefore available to contend for execution, they now dynamically determine their dependence on their specific predicate registers. We now discuss what a predicate to an instruction really is from the point of view of a single instruction now available for execution. From the point of view of any instruction, it can be enabled to execute if control flows to it from one of two categories of sources. First, it may be enabled to execute if control flow is passed to it from its immediate dynamic predecessor instruction. And secondly, control flow can pass to it if it is the target of one or more conditional branch instructions. If either of these conditions become true (dynamically determined at execution time) then the present instruction can be considered enabled to execution. When both of these conditions becomes false, then the instruction becomes disabled to execute. It is that simple, in principle! The remainder of the present scheme essentially manages each of these two categories of possibilities above. In this context, the scheme introduces the idea of a *fall-through predicate* (Pf) and one or more *target predicates* (Pt). These are microarchitectural virtual predicate registers that will govern each of the two categories of instruction enablement already mentioned. Therefore the dynamic setting of these two types of virtual predicate registers during execution is the essence of the present scheme. These two types of predicate registers are evaluated to create the *enabling predicate* (Pe), which is what enables an instruction to execute or not. As will become clear later, time-tag values serve as sorts of addresses for each of the two types

of predicate registers.

7.3.2 Fetch and decode

Instructions are fetched and decoded normally as has been discussed previously in Chapter 3 (Resource Flow execution). When instructions are ready to be dispatched into the execution window, conditional branches are predicted using branch predictors (one per row for example). Whether the outcomes of conditional branches are predicted early in the fetch process of later, such as just before being dispatched, is an implementation option. As with conventional machines, any outcome predictions will serve as a hint to faster convergence towards the proper committed state of the program, but is not required to be correct.

7.3.3 Dispatch

Instructions must be dispatched in a monotonically increasing dynamic program execution order. This is consistent with what has already been discussed in connection with Resource Flow execution (as presented previously in Chapter 3). Note that gaps in dispatch to ISes is indeed allowed. This means that not all ISes need be filled through dispatch in strictly sequential order, only that all instructions that are dispatched are done so in an increasing dynamic program order. An example of a dispatch order that is not allowed would be if the instructions following the taken output path of a conditional branch were dispatched and then the instructions following the fall-through output path of that same conditional branch were then dispatched after some number of the instructions from the taken output path. Note that the reverse order is not necessarily forbidden, but may often be desirable; namely, that instructions following the fall-through output path be dispatched and continuing in dynamic program order even if a join from the taken output path of the preceding conditional branch occurs. This latter situation would indicate the occurrence of a simple single-sided Hammock conditional branch – something that occurs not infrequently – and is certainly allowed with the present scheme.

The decoded instruction, its instruction address, along with the predictions for conditional branches are dispatched to ISes when some become available. Conditional branch instructions are also dispatched with an indication (a single bit) of whether the following instructions to be dispatched follow the fall-through or taken output path of the branch. Indirect jump type instructions can be treated as either a conditional branch with no valid fall-through possibility, or they can be treated as non-branching instructions. The latter case is possible since even enabled instructions at commitment time (something that can happen in this latter case) will not be allowed to actually commit if they are not the proper successor instruction to a preceding control-flow change instruction. All other instructions, including unconditional branches, are treated without distinction

since there is only a single possible output path from them. No assignment of predicate addresses (addresses of the predicate registers governing the execution of the current instruction) occurs at this time as they are rather discovered dynamically as the instructions begin execution. Instructions following conditional branches may be dispatched to start out in an enabled or disabled state (predicated to execute or not) based on the prediction of the conditional branch domain that they are in, but this is not necessary for correct operation. In effect, a random execution predicate can be initially assigned to instructions, as their execution predicate values will likely change anyway as execution proceeds. After a conditional branch instruction has been encountered, instructions following either the fall-through or the taken output path of the conditional can be dispatched – but not both. If the fall-through output path instructions are dispatched after a conditional, joins from either the immediately preceding condition branch or a previous one may, of course, still occur.

7.3.4 Execution

As instructions execute, they will forward predication related information to future program ordered instructions. This is the same in nature as how updated operands are forwarded for registers or memory values. Predication related information is a little bit different and is described in detail later in the chapter. Only relay forwarding is used for the forwarding of predicate information as opposed to something like nullify forwarding (from Chapter 3). Issue stations will snarf predicate forwards similarly to the way that they snarf operand update forwards, by using the time-tag of the forwarding IS in the snarfing logic.

Overflow of branch targets to a single instruction can still occur but these are handled dynamically as encountered. Since instructions were not predicated at instruction dispatch time, the number of control flow paths leading to a single instruction is not known precisely at any given time but is, rather, handled dynamically as the other aspects of the scheme are.

7.3.5 Commitment

As instructions execute, the oldest conditional branch or indirect jump (a *computed GOTO by another name*) in the execution window eventually resolves to its committed state. Upon its last change of outcome indication, it would have forwarded an operand transfer (described later) that changes the enabling predicate of the instructions following it to indicate the correct control-flow path for commitment. This process is chained among all of the conditional branches and indirect jumps in the execution window and therefore all of the instructions eventually determine their enabled state correctly. This convergence towards committed state follows directly analogously from the same convergence with register and memory operands.

For completeness, in variations of this scheme a situation can arise where the instructions

following the taken output path of a conditional branch or the instructions following an indirect jump might be enabled even though the preceding control-flow change instruction resolved not change control flow to their subsequent instructions. However, we note at this point that even if an instruction is enabled to execute through the predication scheme, at no time is it allowed to commit unless it is indeed the correct successor instruction to its preceding conditional branch or indirect jump.

Before more details of the operation of the scheme can be discussed, we need to introduce the necessary IS state that is maintained for the implementation of the scheme. This is done in the next section.

7.4 Issue station predicate state

State is maintained in each issue station to manage the handling of instruction flow predicates. This is similar to the state already maintained within ISes for the management of register and memory operands. The state needed for predicate tracking is a little bit different but makes use of some common principles already used in the management of register and memory operands.

There are two main categories of state maintained in the ISes to facilitate the predication scheme. The first is state that is established when the instruction is first dispatched to the IS and doesn't subsequently change in connection with any predication scheme operations. The second is the state that is dispatched with the IS but which will change dynamically as the predication scheme operates.

The first category of state consists of the following:

- instruction address
- valid fall-through indicator
- IS time-tag

The *instruction address* is the memory address of the instruction being dispatched. It is used in snooping operations for predicates. The *valid fall-through* indicator is a bit that specifies whether the instructions that are dispatched after a conditional branch or an indirect jump follow the fall-through output path or the taken output path. If the fall-through output path instructions are dispatched, the bit is set TRUE, otherwise FALSE. The *IS time-tag* is the time-tag value for the present IS and has been previously described in Chapter 3. It orders the present IS in relation to the preceding and following ISes. None of this state changes due to predication operations after having been dispatched along with the instruction to an IS. Note that the *IS time-tag* does get decremented at commitment time as has been previously presented elsewhere.

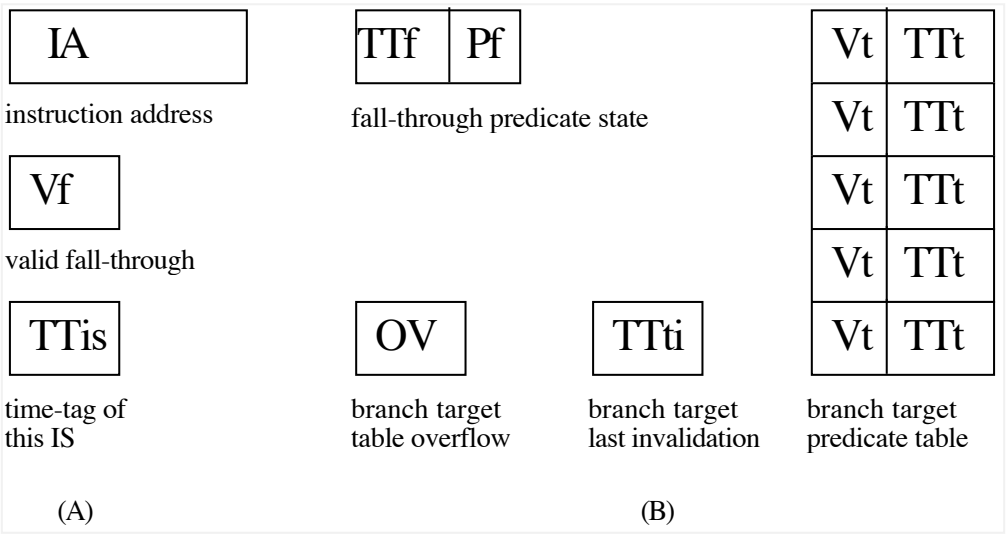

Figure 7.10: *Issue Station predication state information.* The state within each issue station needed for the predication scheme is shown. The branch target predicate table is shown with five entries with each including a target valid bit (Vt) and a target predicate (Pt). Also shown is the the fall-through predicate (Pf) and its time-tag (TTf), an overflow bit (OV), the brach target invalidation time-tag (TTti), the instruction address (IA), the IS time-tag (TTis), and the valid bit indicating whether a fall-through control flow is possible or not (Vf).

The second category of state associated with the predication scheme will change dynamically as the predication scheme operates. Four types of this predicate related state are maintained in each IS. Each type basically consists of one or more registers, either alone or organized into a table structure. These four types of state are:

- fall-through predicate
- branch target predicate table
- branch target overflow bit
- branch target invalidation time-tag value

Figure 7.10 shows the state registers within an issue station that are maintained as part of the predication strategy. The first category of state is shown in part A of the figure (on the left side) and the second category is shown in part B (the right side). For clarity, not shown are any buses or comparison logic that would be used in the snooping process.

A more detailed discussion of this predicate related state is given in the following subsections. The operand transfers that give rise to changes in this new state is outlined in a subsequent section.

7.4.1 Instruction address

This is straight forward. This is the memory address of the instructions dispatched to the IS. This is used by the snooping logic to confirm that an operand transfer applies to the present IS.

7.4.2 Valid fall-through indicator

This is a single bit that is loaded at dispatch time. This bit only applies to conditional branches and indirect jumps. It is not used for any other instructions. This bit indicates whether there are valid instructions dispatched that correspond to the fall-through state of the control-flow change instruction. For indirect jumps, this bit is always set to FALSE since there is no valid fall-through outcome of an indirect jump. An alternative method of handling indirect jumps could leave this bit set to TRUE, but this variation on the scheme is not discussed further in this chapter. For conditional branches this bit is set to FALSE if the instructions dispatched after the branch follow the taken output path of the branch. Otherwise this is set to TRUE for conditional branches. This bit is used in the calculation of the output predicates for conditional branches or indirect jumps. This is described in a later major section.

7.4.3 IS time-tag

This is the time-tag associated with the IS itself. This is identical to the same time-tag discussed in Chapter 3. Although this time-tag doesn't change due to predicate related operations, it does change when a commitment (not other instructions besides the present one) occur in the machine. It is decremented by the number of committed instructions in such a case.

7.4.4 Branch-path fall-through predicate state

The first type of state maintained for predicate tracking is that associated with the fall-through predicate (Pf). The fall-through predicate (a single bit representing whether a branch fall-through region is enabled or not) is similar to the *region predicate* described by Uht et. al. [155] The state associated with the fall-through predicate has two parts and consists of:

- a source time-tag value
- a predicate value

The source time-tag field of the fall-through predicate state is that of the IS that last forwarded a fall-through predicate forwarding transfer (more of these forward operations are described later). This is identical to the last time-tag value maintained for input register operands of the IS. The snarfing condition for the fall-through predicate is identical as that for register operands and occurs

when the source IS time-tag of a fall-through predicate forwarding transfer is less than the current IS's time-tag but greater or equal to the last snarfed value. The last component of state in this group is the 1-bit value of the fall-through predicate itself. A zero value means that the fall-through is disabled, a one value means that it is enabled.

7.4.5 Branch target predicate state

The second major state maintained by each IS for predication is a table of entries where each represents the holding of a *branch target predicate*. The table consists of a number of fixed sized entries and each entry has two parts. The idea of a branch target predicate is a virtual one since its presence and value (always being TRUE) is indicated by a valid table entry. The lack of an entry for a target predicate with a particular time-tag (TTt) value is taken as unknown (either TRUE or FALSE). The parts of state for each entry consists of:

- a valid bit
- a source time-tag value

The valid bit (Vt) indicates that the entry is used and that the associated time-tag value is valid. Any valid entry indicates that the current instruction is the possible target of a previous conditional branch and that the current instruction is therefore currently predicated to execute (is enabled). This further means that the output fall-through predicate from this instruction should be set to TRUE for non-branch instructions. For conditional branch instructions, an additional factor is involved for determining the proper output fall-through predicate value. The exact calculation of the instruction execution predicate and its output predicate for both non-branch and conditional branching instructions is discussed in a subsequent section. It should be further noted that the time-tag value (TTt) in a branch target predicate is effectively the address of the predicate. The presence of a valid entry in the table (valid bit is set) means that a branch target predicate has been previously received by this IS. The predicate itself (if present in one of these state table entries) is implicitly TRUE (and therefore enabling for the associated instruction) and can be thought of being sourced from the conditional branch that originated it. The time-tag value is the address of the IS holding that originating conditional branch. Although no actual bit is allocated to hold the branch target predicate, one can still think of having received one, with its value being TRUE. This abstract notion of holding a branch target predicate corresponds with the *cancelling predicate* described within Uht's scheme [155] and serves the same sort of a purpose. The branch target predicate was termed a cancelling predicate in the previous work because it served to *cancel* the negative (instruction disabling) effect of a FALSE fall-through predicate. In the present work, this

same notion of a cancelling predicate is more naturally expressed as serving as an enabling of the present instruction, regardless of the state of any fall-through predicates.

Any number of table entries are possible but the number should roughly correspond to the number of conditional branches that are likely to target a single instruction on the average given the size (in main-line ISes) of any given execution window. Program characterization can be used to pick a good value for the number of entries. When more branch target predicates are received than there are available entries, an overflow mechanism is invoked. Predicate-only issue stations are neither assigned at dispatch time nor are they used in the scheme. Rather, a dynamic overflow scheme is used to handle these circumstances. Invalid entries are available for storing time-tags for new branch target matches to the current instruction.

7.4.6 Overflow indication

Next, there is maintained an overflow bit (OV) for those cases when a given instruction is the target of more conditional branches than there are branch target predicate entries in the table just described. When the overflow bit is clear, it indicates that an overflow condition is not in effect for the current IS. If the bit is set, it indicates that an overflow of branch target predicate matches with the current instruction address has occurred. Since the existence of branch target predicates for the current instruction always indicates that the current instruction is predicated to execute, no ambiguity of the predication status is manifested until all existing branch target predicates are invalidated. The exact details of the handling of an overflow is discussed later.

7.4.7 Invalidation time-tag state

Finally, each IS maintains a state register that holds a time-tag value (TTti) that further governs the management of the branch target predicate table. This time-tag value is used to prevent the snarfing of branch target predicate broadcasts that are not applicable to the current issue station. This state register is (or may be) updated with the latest sourcing IS time-tag value that was associated with a branch target predicate invalidation forwarding transfer. This is very analogous to holding the latest time-tag value snarfed for a register operand but it is applied to invalidation forwarding transfers rather than the more familiar store-update forwarding transfers associated with register and memory handling.

7.5 General operation

As instructions execute they forward any significant changes in predicate information to instructions lying in the program-ordered future (those with larger valued time-tags). The later instructions (in

program order) use the forwarded information to determine their branch domains and whether they are the target of one or more previous branches. A number of predicate operand transfers are defined to facilitate the transfer of predicate related information among the ISes. These operand transfers are new and are in addition to the previous operand transfer that were first introduced in Chapter 3. These are also specific to this predication scheme. Another or future predication scheme would likely have its own set of predicate related operand transfers. The following subsections discuss the operand transfers needed to support our newly introduced predication scheme. A further discussion of the general operation of the predication scheme follows the discussion of the various operand transfers.

7.5.1 Predicate related operand transfers

There are a total of four new operand transfer operations that we introduce to support our new predication scheme. Three of these are forwarding requests, the fourth being a backwarding request. The predicate-related forwarding operand transfers are entitled:

- fall-through predicate
- target predicate
- target predicate invalidation

The predicate-related backwarding operand transfer is entitled:

- resend predicates

The operand transfer types (or opcodes) are implied when discussing the contents of each of the operand transfers, however the transfer type is indeed always present on the actual hardware transfer buses. Each of these operand transfers is now discussed in turn.

Fall-through Predicate operand transfer

One type of predicate forwarding transfer is generated from non-branch instructions. This predicate forwarding transfer type forwards a fall-through predicate value similar to a register operand store-update forwarding transfer. When a non-branch instruction forwards predicate information, it will forward only a fall-through predicate. The following is forwarded in this type of forwarding transfer:

- the sourcing IS time-tag value
- the fall-through predicate value

The snooping ISes in the program ordered future will possibly snarf this predicate value according to normal snarfing rules for the comparison of time-tags that are already used for register snarfing. Snarfing ISes will update their fall-through predicate state to reflect the latest TT value and the new value for the fall-through predicate.

Branch-Target Predicate operand transfer

Another type of predicate forwarding transfer is generated by conditional branch instructions and indirect jumps. This type of predicate forwarding transfer forwards a fall-through predicate value as well as a branch target predicate. More information is contained in this type of predicate forwarding transfer as compared with the simpler fall-through predicate forwarding transfer already described. The fields associated with this forwarding transfer include:

- the sourcing IS time-tag value
- the branch target address
- the output fall-through predicate value
- the output branch target predicate value

Snooping ISes, upon seeing a branch target predicate forwarding transfer, will snarf and update their fall-through predicate state the same as if a fall-through predicate forwarding transfer (from a non-branch instruction) was forwarded. However additional work is done also. The snooping IS will compare its instruction address (which it received at instruction dispatch time) with the branch target address that is forwarded by the branch instruction. If there is a match on the branch target address, the snooping IS will also look to see if the branch is predicated taken. If the branch target predicate is true, then that indicates that instruction flow is currently predicted to come through the taken path. In this circumstance, the snarfing IS allocates a previously unused branch target predicate state register for holding the TT value of the originating IS with the conditional branch.

Note that both a fall-through predicate and a branch target predicate is supplied in this operand transfer. This is because it is possible for both of these to be FALSE. Indirect jumps will always generate this operand transfer with a fall-through predicate of FALSE. Conditional branch instructions will generate this transfer with a fall-through predicate of FALSE if its fall-through indicator bit is FALSE (set upon dispatch). This condition corresponds to when subsequently dispatched instructions followed the taken output path of the conditional branch. For indirect jumps, the equivalent of the taken output path is always followed (since there is no possible fall-through output path).

Branch-Target Invalidation operand transfer

Each branch target predicate forwarding transfer can only logically be used by at most one instruction located in the program ordered future of the conditional branch that originated the forwarding transfer. A means is needed to ensure that only one instruction (the true target of the conditional branch) makes use of a branch target predicate forwarding transfer. More than one instruction in the program ordered future from the conditional branch may match on the branch target address. This can occur when (for example) the target of a branch is located within a loop in the program code. Since several instructions (for example, one from each iteration of a loop) can have the same instruction address, some means needs to be provided to distinguish the first of these (in program ordered time) from the remaining (subsequent) ones. This situation is handled by the IS holding the instruction that is the target of a conditional branch. This IS may be either the actual target of the branch (the first matching one after the conditional branch) or a subsequent instruction. In either case the same action is taken. Such an IS forwards a special type of forwarding transfer that will invalidate the effects of the previous branch target predicate forwarding transfer. This forwarding transfer consists of the following fields:

- the sourcing IS time-tag value
- the branch target instruction address
- the time-tag of a branch target predicate that is to be invalidated

The sourcing IS time-tag value is used in the snooping determination as is typical with other (register and memory) forwarding transfers. The branch target address is also used to see if it matches with the instruction address in the snooping IS. When a match is determined, a search is made in the branch target predicate table for a matching time-tag for the IS holding the original conditional branch. This latter search can actually occur in parallel with the branch target comparison but that is an implementation issue. If an entry is found, it is deleted (marked as invalid) and that entry is available for reuse. Also, if an entry is found, the time-tag state register that holds the last invalidation time-tag value, maintained within the IS, is updated to reflect the sourcing IS's time-tag value. This invalidation time-tag value can be used to avoid false branch target predicate matches in the future. Once an invalidation time-tag value is acquired, a future branch target predicate forwarding transfer is needed to be sourced from an IS with a time-tag value that is at least as large as the invalidation time-tag value. This use is correctly forcing an IS to avoid accepting branch target predicates that actually belong to the correct target of the branch (which is not within the present IS). This feature is not strictly needed, as the whole scheme continues to work properly without maintaining this invalidation time-tag state. But it does serve to reduce some unnecessary flipping of the enabling execution predicate for false branch target matching instructions.

Resend Predicates operand transfer

This is the only backwarding operand transfer needed to support the scheme. This forwarding transfer consists of the following fields:

- the current instruction address

This type of operand transfer is sent on the backwarding operand transfer fabric and is used in the case when an IS enters into an ambiguous state associated with its branch-target predicate table. Specifically, an ambiguous IS state is entered when the branch-target predicate table overflows in any IS. In this case, this *resend predicates* operand transfer type is used to resolve the ambiguity. Upon receipt of this operand transfer type, all ISes containing conditional branch instructions resend their output predicate state. This serves to initiate a chain reaction of predicate forwards that resolves the ambiguous state of any IS in such a state.

7.5.2 Detailed operation

Given a fall-through predicate (Pf) and any branch target enabling predicates (Pt) from the branch target predicate table, the execution predicate (Pe) of the IS (its enabling predicate) is computed as:

$$Pe = Pf, input + (\text{any branch target predicates} - \text{Pt})$$

Similarly, a new output fall-through predicate (Pf) for a non-conditional branch instruction is computed by the snarfing IS as follows:

$$Pf, output = Pf, input + (\text{any branch target predicates} - \text{Pt})$$

Note that this is the same as:

$$Pf, output = Pe$$

as would be expected since if a non-branching instruction is ever enabled for execution, any instructions following it up to and including a subsequent conditional branch are also automatically enabled for execution. If the output fall-through predicate changes from its previous value, it is forwarded as expected. As expected, the output predicates from conditional branch instructions are computed according to:

$$Pf, output = Pe * Vf * (\text{branch indicating not-taken})$$

and

$$Pt, output = Pe * (\text{branch indicating taken})$$

The first of these is the output fall-through predicate (the output predicate for the not-taken output path from the branch). Note the inclusion of the fall-through indicator bit in the computation. That bit prevents the generation of a TRUE fall-through output predicates (Pf) for either those instructions that cannot generate such a condition (an indirect jump) or for conditional branches where the subsequently dispatches instructions followed the taken output path of the branch. The second computation above is the Branch Target Predicate (Pt) and is for the target of the branch. Both output predicate values are forwarded using a Branch Target Predicate forwarding transfer as previously discussed.

The significance of forwarding the branch target instruction address rather than the target time-tag value (an idea previously considered) is to handle the situation where the target of a branch does not lie the same number of ISes forward in program ordered time as the branch target address alone might indicate. This fact defeats the feasibility of using the combination of the present instruction address and the target instruction address (both of these being available) along with the present instruction time-tag in order to compare the time-tag of the target instruction. If a computation of the target instruction time-tag was possible, then presumably it could be used to direct a predicate transfer operation directly to the target instruction by placing the target time-tag value in the predicate transfer itself. But this isn't possible in general while also allowing for an entirely flexible dispatch order algorithm (which we certainly want to handle). A change in the instruction dispatch order into the execution window from a static program order to an order where instructions are dispatched following the taken output path of a branch may have occurred. This obviously causes the target instructions on the taken output path of a conditional branch to lie in the execution window immediately after the conditional branch dynamically (in program order) preceding them. This renders the relative computation of a target TT value from the difference of the instruction addresses of the two instructions and the present time-tag to be of no use. We note here that a variety of dispatch order decisions could occur following a conditional branch (including encountering additional or unconditional branches) that can lead to situations where a branch target instruction may indeed be located anywhere within the execution window. A situation like this makes the target instruction's time-tag unknowable by any information present by the originating conditional branch. Rather, by having conditional branch instructions broadcast the absolute target instruction address during predicate operand transfers, the possible target instruction can snarf on an instruction address match regardless of how many ISes into program ordered future it may lie within the execution window.

When a branch instruction computes a change in its output branch target predicate such that it becomes false (the branch is no longer predicted to be taken or it is no longer predicated to execute itself), it will perform a predicate forward operation. Snooping ISes will again match on the target address but will also search to see if that branch target predicate from the originating branch

instruction was previously recorded. If it was previously recorded, the branch target predicate address state register entry that was occupied becomes unoccupied (its VALID bit is set to FALSE). This amounts to the loss of an enabling input to its execution predicate computation and its output fall-through predicate value. If the output predicate fall-through value changes, this instruction will in turn perform its own predicate forward operation. Of course, the target of a branch instruction can itself be a branch instruction and this is handled in the expected straight forward manner (as in the previous scheme also).

No ordering of any kind is needed for the forwarding of predicate transfers, whether it be from a non-branch instruction or from a branch instruction. This latter feature is a critically important one because the current predicate forwarding bus mechanisms, when combined with the existing grouping of ISes into Sharing Groups, cannot necessarily guarantee the order of predicate forward operations. The order of predicate forward operations may be mixed up due to additional structural hazards in the machine such as the priority and queue delay for the use of a Processing Element when one may be needed for the execution of a branch that may require some non-trivial computation. In any event, it is certainly advantageous not to require a strict real-time ordering of predicate forward operations.

An apparent difficulty with the new scheme occurs when the target of a branch already has all of its architected (machine configured) branch target predicate-address state registers used and then a match occurs with a new originating branch instruction that was not previously recorded in the IS. What we would like to do is to allocate an additional predicate address state register to hold the new predicate source TT value, but this is not possible since all of the registers are used up. Rather we can either ignore the new branch match condition or we can replace an existing branch predicate address with the new one. In either case, the overflow bit is set in the IS state. The replacement policy for handling overflow branch target matches flexible and differing policies may be better for different workloads. More on possible replacement strategies is discussed in a subsequent section. We have to set the overflow bit because we need to track that an overflow occurred. This is necessary so that after all existing branch target predicates become false, due to subsequent forward operations, and when the fall-through predicate of the current IS also goes false, we need to know that there is still an ambiguity about whether the current IS is predicated to execute or not. The ambiguity in the IS is resolved through the initiation of the *resend predicate* backwarding transfer. This is the fourth of the above predicate operand transfers and directs that all conditional branch instructions that have targets matching the instruction address of the present IS resend a predicate forwarding transfer of an appropriate type (one of the first two types) so that any branch target matches for those branches that are predicated to execute, and to execute taken, can be snarfed by the ambiguous IS. At least one backwarding *resend predicate* request would have to be made for any IS in an ambiguous state before the ambiguous IS could be allowed to commit.

7.6 Branch target predicate table entry replacement policies

Some thought can now be given to the replacement policy for storing matching branch target predicate addresses. We obviously would want to minimize the likelihood of having as IS ever reaching an ambiguous state. The likelihood of an IS reaching an ambiguous state is already probably quite low but always remains non-zero if the number of configured branch target predicate address storage registers in an IS is less than the IS window size (same as the previous scheme we are enhancing). The likelihood of an IS reaching an ambiguous state will only occur after both overflow occurs and when its fall-through predicate goes false. This is somewhat less than the probability of some number of preceding branches all predicting that they are both being executed and are also being predicted as taken, and when all of those branches target a single IS. This may indeed be a very small probability. Simulations can show the exact likelihoods involved with actual benchmark programs codes. Replacement policies that have been thought of include:

- ignore the latest match and keep existing entries
- replace a branch target predicate address such that the latest time-tag values are retained
- replace a branch target predicate address such that the earliest time-tag values are retained
- maintain an age associated with a branch target predicate address (updated each clock for example) and only replace the oldest aged entry
- maintain an age and replace the youngest aged entry
- some combination of the above

7.7 Summary

We have presented the overview of a new microarchitectural predication scheme. Our new scheme, being microarchitectural, allows for both its use with legacy ISAs as well as possibly improved execution performance due to the fact that all instructions are predicated rather than just those that can be selected for predication through a programmer or a compiler. Our new predication scheme performs dynamic discovery of predication dependencies at execution time. This is an improvement on the previous scheme that determined its predication dependencies at dispatch time, requiring a centralized hardware predicate tracking buffer. Our new scheme also allows for the elimination of any centralized hardware resource and the buses associated with that resource that had to span the entire physical size of machine (to reach all issue stations). Our scheme therefore allows for better physical machine scalability than the previous scheme did. For this

reason, our new predication scheme is something of an enabling technology for implementing a physically scalable machine designed using the philosophy of resource flow computing. As will be seen, this scalable attribute of our newly introduced scheme is an enabling factor for a generally scalable microarchitecture presented later (Chapter 10).

Chapter 8

Operand Forwarding and Filter Units

In Chapter 3 we introduced the idea of the Resource Flow execution computing model. In Chapter 4 we introduced one possible microarchitecture using the Resource Flow model. That microarchitecture, called OpTiFlow (Operand Time Flow), was not itself suited for large physical scalability as it was. However, the hardware components used within OpTiFlow, specifically the issue stations, the register file, and the load-store queue, were designed for and operate in a way that allows them to be decoupled from each other in terms of the number of clock periods that can elapse for an operand transferring from one of those components to another. This inherent time decoupling between hardware components operating under the Resource Flow regime suggests that additional elements can be inserted within the operand flow path from one component to another without requiring any design changes to the existing components of a Resource Flow microarchitecture, with the OpTiFlow microarchitecture being one example. In this chapter we examine ways to take advantage of the loose time coupling between components making up a Resource Flow microarchitecture. Our goal is to achieve physical scalability within a Resource Flow microarchitecture and we present here a proposal using that as the starting assumption.

The remainder of this chapter is organized as follows. We first briefly examine some of the problems associated with implementing physical scalability within a processor microarchitecture. Many of these problems are inherent to any large system of logic circuits but our focus will be more towards how to overcome them for Resource Flow microarchitectures. We then introduce simple but adequate ways to overcome some of the scalability problems. Next we introduce enhanced capabilities (what we will term *operand filtering*) that might be applied to the hardware already needed to solve some of the scalability problems. Finally we summarize.

8.1 Problems with physical scalability

The principal problem with physical scalability has to do with the need to interconnect hardware components. An important way to interconnect machine components is by placing the various components on buses. It is therefore the problems that are introduced by attempts to physically scale buses that will be our primary concern. As is the case with many processor microarchitectures, a number of buses are used to transport logical values from one component to another. Buses might be simple point-to-point connections or they may be multipoint connections of a few varieties (point-to-multipoint, multipoint-to-point, and multipoint-to-multipoint). Additionally buses may operate in strictly a unidirectional mode or may operate bidirectionally. The idea of the point-to-point bus is the degenerate case of the idea of a bus in general. And a unidirectional point-to-point bus is nothing more than just a simple interconnection of a source of logic signals with its one sink for those same signals (we consider each bus to possibly be a collection of parallel individual logic signals). But it is the utility of the other ways that buses can be used that make them so valuable for a variety of logic transport tasks within general logic systems as well as computer microarchitectures. For this reason, buses that operate in more than just a point-to-point capacity make up a significant portion of many logic machines and microarchitectures. For our purposes, we will assume the worst of these cases, being that of multipoint-to-multipoint buses, and examine ways to deal with that for achieving scalability.

In the following subsections we examine some of the problems associated with buses when trying to scale (extend) them. The first and most evident problem with scaling of buses is that of having indefinitely long bus lengths. Another problem is that extending buses logically through the use of bus repeaters may (and generally does) cause timing problems. The next problem we will examine is that of managing pipeline stages resulting from the splitting of buses into pipelined segments. And finally we will look at the issue of bus bandwidth issues associated with scalability. These problems are discussed in turn below.

8.1.1 Bus length

One way to scale a system of logic to larger sizes is to add logic components in a like manner as existing identical components already within the original system. However, when these components are sharing all one or more common buses, this would require the extension of the bus to longer physical lengths to accommodate the larger physical space taken up by the additional components. This situation can be illustrated by examining how we might extend the OpTiFlow microarchitecture to have a larger number of components. For reference, the high-level block diagram of the OpTiFlow microarchitecture is shown in Figure 8.1. For purposes of illustration, assume that we wanted to add additional issue stations to the microarchitecture while retaining the existing

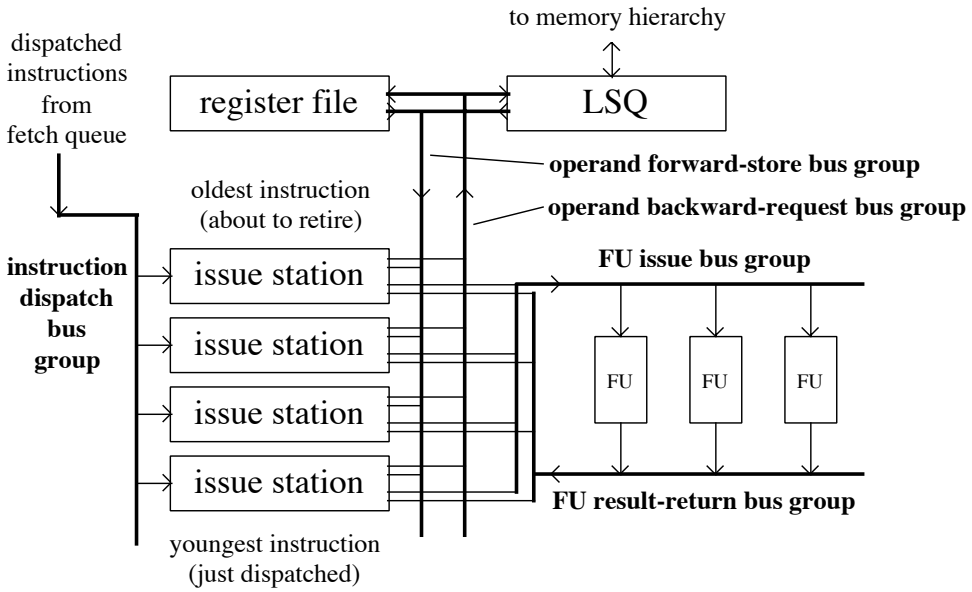

Figure 8.1: *Simplified block diagram of the OpTiFlow microarchitecture.* Shown is a simplified block diagram of the OpTiFlow microarchitecture with some of its primary buses included. For example, the issue stations (shown along the leftward part of the diagram, are all connected to the vertical operand transfer buses of two types: forwarding and backwarding. Additional issue stations cannot be simply added to this microarchitecture indefinitely using the existing bus arrangement without severely compromising the signal integrity and propagation delay of signals on the operand transfer buses.

simple operand transfer bus arrangement, one set of buses interconnecting all issue stations (shown vertically in the figure interconnecting all issue stations). We would not be able to add additional issue stations indefinitely without eventually degrading the signal propagation characteristics on the operand transfer buses. Simply extending buses to longer lengths has a number of critical problems associated with it.

The first of these problems is the additional propagation delay introduced by the longer bus. This itself has two subcomponents. The first is the fact that given a constant propagation speed, a longer bus will require a longer period of time for a signal to propagate down its length. The second subcomponent that leads to increased propagation delay is the problems associated with additional loading on the bus. The additional loading itself comes in two forms. The first is from the input loads of additional components being added to the bus. The second is from additional loading being present from the additional stray capacitance of the bus due to its longer length (we are assuming the bus is laid out in typical fashion within silicon). Both forms of additional loading effectively increases the rise time of the bus signal and therefore presents an additional delay due to the retardation of the signal reaching the logic switching threshold of the various logic receivers. Further, and not to put too fine a point on this problem, when the additional loading on the bus is finely enough distributed in the spatial dimension with respect to the rise time of the bus signal, the effective permittivity of the bus is increased causing a decrease in its speed of signal propagation. This is a wave-propagation phenomenon of the situation but can occur in actual circumstances. Similarly, with the increase in effective permittivity of a bus also comes a decrease in its effective bus impendence. This in turn can lead to a situation where the propagated signal cannot cause an incident wave switch at the logic receivers due to the fact that a large enough voltage to cross the receiver's logic switching threshold cannot be created on the bus due to an inadequately powered logic driver. The original system (before any buses were simply extended) was presumably designed to accommodate the necessary logic propagation delay incurred by all of its buses. However, it would not be acceptable to simply assume that excess timing margin was available in the original design to accommodate increased propagation delays due to increased bus lengths. In reality most well designed systems do not generally have excessive timing margins within them since that would have constituted something of a waste of capability of the technology. Further, it must be remembered that although a logic system may appear to have a large excess timing margin at room temperature and average silicon technology processing transistor transconductance gains, it will not have this excess margin at all of its intended operating extremes of temperature and technology processing.

A second major problem associated with employing longer physical bus lengths has to do with the possibly cross coupling of logic signals from one bus lead to another, or many others. The physical extension of a bus generally increases the cross coupling of the bus (or some of its constituent bus leads) to adjacent physical buses within the silicon layout. When cross coupling of

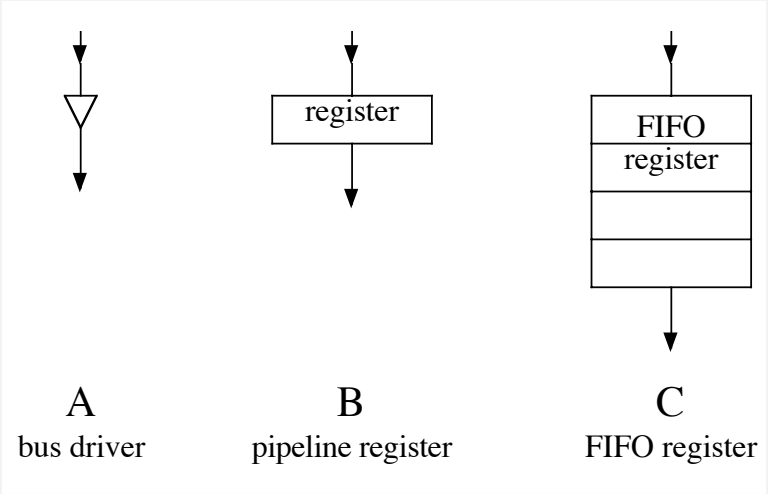

Figure 8.2: *Three means to extend a bus.* Shown are three different means that can be used to extend a bus. The one in part A (left) is a simple logic driver. It electrically regenerates the bus signals but adds propagation delay. The second (part B, middle) is a register. It combines the features of the driver (part A) with a clocked register. This effectively introduces a pipeline stage. The third (part C, right) is a set of registers organized in a First-In-First-Out (FIFO) way. It provides electrical regeneration but also introduces a variable amount of clock delays.

signals becomes too high, false logic signals can be inadvertently and pseudo randomly generated that can wreak havoc for the logic system. Although issues associated with cross coupling might be resolved using alternative silicon layouts, a common way to deal with this is to simply shorten the length of buses overall.

The technology problems associated with increased bus lengths cannot be solved through some technology means alone. So the problem of any increased propagation delays becomes either one at the semantic level of logic or possibly even a microarchitectural concern, rather than just a technology concern. The solution to this problem is to not necessarily physically increase the length of buses but to rather handle them at a higher level sematic level. But solutions in either logic or microarchitecture may also introduce design problems, albeit at the higher semantic levels.

8.1.2 Electrical signal regeneration

One apparently simple way to handle the problem of long buses is to limit their length by breaking up long buses into shorter length segments by using a bus logic driver circuit. This circuit electrically regenerates the logic signals on its input to its output. This type of circuit is shown in part A of Figure 8.2. For those cases where the bus is bidirectional (logic signals are expected to flow in both directions at different times), two such drivers can be used in parallel but in placed to drive in opposite directions. In this case a control signal is needed to indicate the enabled direction of logic

transfer. While a fresh logic signal is created for the next section of the logical bus from the output of the driver, a propagation delay is also introduced. If the original bus had a timing budget for the propagation of its signals from one end to the other, the introduction of a logic driver would cause a timing violation (missing set-up time to the next registered logic component) unless excess timing margin was present. This is not generally the case. We assume that all bus use is synchronized with a system clock and that each transfer occurs within a single clock period. For this reason, using simple logic drivers to extend buses is not often useful for systems with tight timing constraints. Our system, represented for example by the OpTiFlow microarchitecture one such system. And for this reason, using simply logic driver circuits is not possible. Other alternatives are required.

8.1.3 Register pipelining

A better way to handle the timing problems introduced with a logic driver circuit is to add a register to the driver circuit so that all timing can be kept synchronized to the system clock. A diagram of this sort of arrangement is shown in part B of Figure 8.2. Although the driver circuit part of the whole circuit is not shown (by convention) it is present on the output of the register. The register is also clocked by some clock signal that determines when the input signals of the register are transferred into the register and onto the output driver of it.

The introduction of a clock period delay in the availability of the output of the register would need to be accommodated for by the receiving machine components. The receivers would need to expect the data a clock period later than if they were on the input side of the register. But that is not the only problem with this arrangement. If we tried to apply the idea of using a clocked registered repeater for use in extending the operand transfer buses in the OpTiFlow microarchitecture (reference Figure 8.1, we would have something similar to that depicted in Figure 8.3. In this arrangement, the operand transfer buses in each direction (forwarding and backwarding) are effectively segmented through the use of a clocked register. Although this simple register scheme is adequate for limiting signal propagation delays for all of the clock-synchronous circuits it has a somewhat severe operational restriction associated with it in the context of the our application (extending operand transfer buses).. When an operand is put onto an operand transfer bus, it will be latched by all receiving operand related components (LSQ, register file, issue stations) and also by any operand bus repeater unit. All components latching an operand off of an operand transfer bus must be ready to receive a new operand in the very next clock period. This is true for the operand forwarding unit also. But since it only has space to store at most one operand, unless the registered operand within the unit can be vacated in the very next clock period, a stall will have to be initiated for the bus on the receiving (bus input) side of the unit. This stall will prevent a bus transaction from proceeding until the stalling condition is cleared. The reason that the stall might

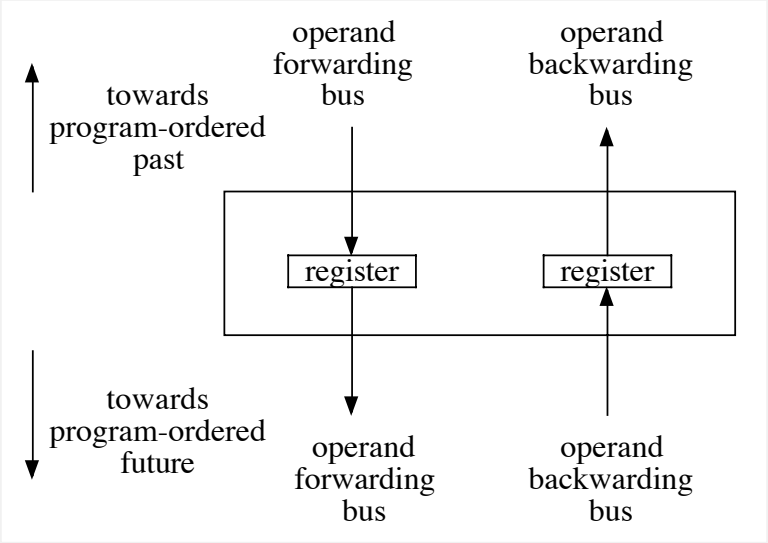

Figure 8.3: *A possible operand transfer bus repeater using registers.* Shown is a possible operand bus repeater circuit using simple registers. One register is used for each of the two buses (or sets of buses) used for each direction of operand transfers: forward and backward. This arrangement will not work when the output bus is also a multimaster bus, which is generally the case.

occur is due to the need for the operand forwarding unit to arbitrate for and win the mastership of the multimaster operand transfer bus on its output side. In practice, an arbitration for the output side bus and the necessary vacating of the register contents cannot occur at the rate of one per clock, which is what would be needed for transfers occurring in each clock period (an operational requirement). For this reason, this sort of scheme (a simple registered bus repeaters) is not usable for our purposes. At a minimum, more than a single operand must be able to be stored within the operand bus repeater unit in order to allow time for bus arbitration to occur on the output side of the unit. A repeater unit with more storage is indeed adequate for proper minimal operation and this is introduced in a subsequent section.

8.1.4 Bandwidth congestion

A more subtle problem with the use of multimaster buses (as we have in the OpTiFlow microarchitecture with its forwarding and backwarding operand buses) is the fact that as components are added to a common bus, the utilization of the bus (those clock periods that have logic transfers in progress) will likely increase. This is due to the fact that the amount of bus bandwidth available on a given bus remains the same while the competition for bus use is increased from the additional components arbitrating for a bus clock transfer period.

If we could segment a long logical bus into multiple smaller segments, each of which could carry

on bus transactions in parallel, the problems of bus bandwidth congestion could be mitigated if the addition of more hardware components also was accompanied by the additional of more independent logical bus segments. This is indeed what we will do for the OpTiFlow microarchitecture. In the following sections we will introduce those methods of extending logic buses that are compatible with use on a Resource Flow microarchitecture such as OpTiFlow.

8.2 Operand forwarding in Resource Flow execution

In this section we introduce a bus extension idea that is compatible with our Resource Flow execution model (the OpTiFlow microarchitecture being one example). Of course the electrical problems associated with long bus lengths requires that buses be broken up into logical segments. In the case of a Resource Flow microarchitecture, the most significant buses that prevent physical scalability are the operand transfer buses. There are two varieties of these, forwarding and backwarding (reference Figure 8.1), and the bus repeater presented here will work with both of these. As we saw in the last major section, significant problems are present with both a non-registered and a simple registered bus repeater circuit. A clock registered design for the bus repeater is needed but also there needs to be space within the repeater unit to buffer up more than a single operand in order to allow time to arbitrate for the output bus (which is itself a multimaster bus). A first-in-first-out (FIFO) register buffer fulfills this requirement. An example of a registered FIFO being used to extend a bus is shown in Figure 8.2 part C. When this idea of using a FIFO is practically applied to our application with the forwarding and backwarding operand transfer buses, a design similar to that depicted in Figure 8.4 is created. We term this combination of registered FIFOs for the operand buses (forwarding and backwarding) an *operand forwarding unit*. Although the name suggests only operand forwarding, operand backwarding requests are also handled by it. As shown in the figure, the operand forwarding unit interfaces to four different buses. These are the forwarding and backwarding buses on the side of the forwarding unit facing towards the program-ordered past when considered in the context of the microarchitecture, and the forwarding and backwarding buses on the unit facing towards program-ordered future within the overall microarchitecture. Further, it may also be the case (and as we have allowed for throughout) that each of these four buses may be in themselves a set of multiple parallel buses which simple multiple the overall available bus bandwidth by the number of parallel buses employed. Still further arrangements of multiplying operand buses are possible, other than simple parallel bus bundles, but those arrangements are beyond the scope of our present discussion. Also, it is noteworthy that this operand forwarding unit described can be used to forward any type of operand that may be present on the operand transfer buses. This unit is blind to the particular type of operand since it merely forwards it without regard for its type.

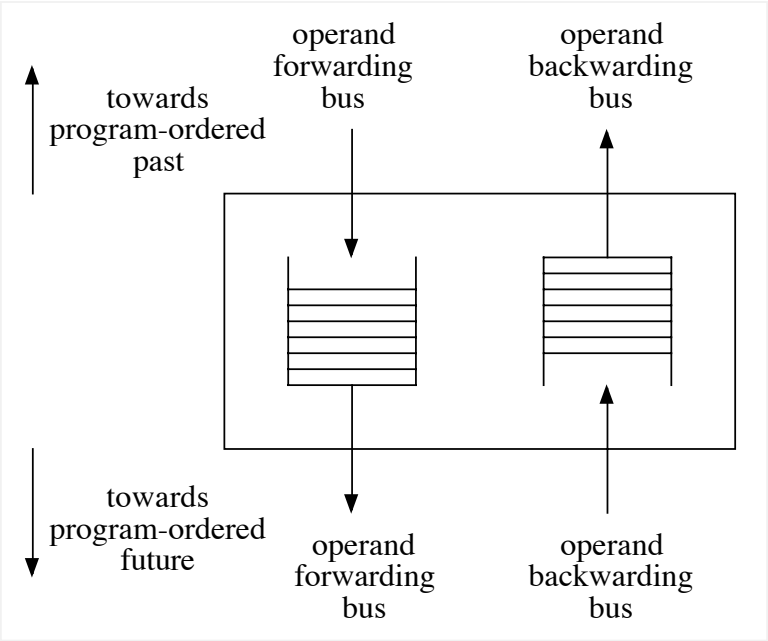

Figure 8.4: *The simplest working operand bus repeater design for OpTiFlow.* Shown is the simplest possible operand transfer bus repeater unit that can be used for extending the operand transfer buses within a Resource Flow execution microarchitecture. One registered FIFO is used for each direction of the operand transfer buses (forwarding and backwarding).

This scheme works within the Resource Flow execution model because of the way that operands are passed from older executing instructions to younger executing instructions. Specifically, there can be an arbitrary delay between when an operand is forwarded and when it is received by subsequent issue stations. This is also true for operand backwarding requests. It is currently assumed that all components in the system are still clocked the same (clock synchronized throughout) but even this is not a requirement of how operands can be transferred among issue stations. In theory, groups of issue stations could be clocked together and there would only need to be some clock synchronization mechanism at operand transfer bus repeater boundaries.

With the addition of the registered FIFOs, enough storage is now available for operands so that arbitration can occur for the output side bus without necessarily stalling the input side bus. However, there are still undesirable operating properties even with this FIFO repeater design. In this design the entire bus bandwidth of the input side buses (each side of the repeater has an input bus) must still be transferred to the corresponding output bus. For highly utilized operand transfer buses, this condition will still cause a large degree of bus stalls. Rather we would like to find a way to limit stalls while also allowing the bus segments on each side of the bus repeater to operate substantially independently with local transfer traffic. This is exactly what we explore in the next section.

8.3 The operand filtering idea

The problem with the operand forwarding unit of the last section (the use of simple registered FIFOs) is that it requires that the buses connected to it (on both sides of the repeater unit) be substantially underutilized so as to not always be in a state where bus stalls are forced due to the FIFOs being filled up with operands. Unlike with the situation of traffic congestion in many packet date networks, the dropping of operands is not a possible option as a way to deal with operand storage exhaustion within the repeater units. However, maybe we can take advantage of some of the properties of the operands being received by the forwarding unit to determine if one or more of the operands do not need to be forwarded onto the connected output operand buses. If some operands being forwarded, or requests for operands being backwarded, can be either ignored or satisfied within the forwarding unit itself, this would create a situation where bus transfer bandwidth would be freed up on the connected adjacent buses for additional transfers. This would correspond to better utilization of all buses and should result in an overall performance improvement for a given bus bandwidth. To accomplish this goal of limiting the need to forward or backward operand transfers, we will also consider the type of the operand involved (register, memory, or predicate) and the direction of the transfer under consideration (either forward or backward in program-ordered time). When multiple forwarding units are used that distinguish what types of operands

they can handle, they can still share common operand transfer buses (buses that carry all types of operands) by simply arranging them in parallel with each other and connecting to the same buses as each other. We will look at these possibilities in the subsequent sections.

8.3.1 Memory operand filter

After having considered the idea of using FIFOs within the operand forwarding unit, perhaps additional forms of operand storage might also be possible and useful. But how this new operand storage might be integrated into the operand forwarding unit and what form it might take has to be carefully considered. In the general case, FIFOs are still needed on the inputs from and outputs to all connected buses, so these structures need to be retained in any future designs. In the operand forwarding unit of the last section, a total of only two FIFOs were used: one in each direction. But each of those FIFOs could actually be considered two sub-FIFOs in series (or four sub-FIFOs in total). If we follow through with the idea of separating the input sub-FIFO from the output sub-FIFO, some form of additional operand storage might be placed between them. An arrangement of this very sort is shown in Figure 8.5. In the figure, the two FIFOs that were formerly used for each direction of transfer are now divided into four FIFOs. Each of these serves a single input or output bus interface. An operand cache has now also been added between the FIFOs. This cache is similar to a conventional memory cache except that now a whole operand (a value along with all of its metadata) is stored rather than a simple memory value. In the simplest case, operands within the cache are indexed using the metadata of the operand consisting of the memory address and the time-tag.

The operand cache is used in the following ways. When a new operand arrives from the program-ordered past (from older instructions within the execution window) an update of the cache is done with the new operand. If the operand was not previously located within the cache, the cache is updated by possibly evicting an existing cache entry. Also, if the operand was not previously present, it is also forwarded to the output operand transfer bus towards program-ordered future. If the operand was found to be in the cache, an optimization is possible here by not forwarding the operand to the output bus. Operands do not need to be forwarding if the newly arrived data value is equal to the existing data value in the cache. For backwarding requests, when a new backwarding request arrives from the program-ordered future (backwarding requests travel from program-ordered future towards program-ordered past) a lookup is first done to the cache. If the operand is found to be in the cache, it is retrieved from the cache and placed in the output FIFO for the output bus going to program-ordered future. In this case, the backwarding request does not have to be backwarded further. This represents an optimization and saves from bandwidth on the output backwarding bus (going toward program-ordered past). If the operand corresponding

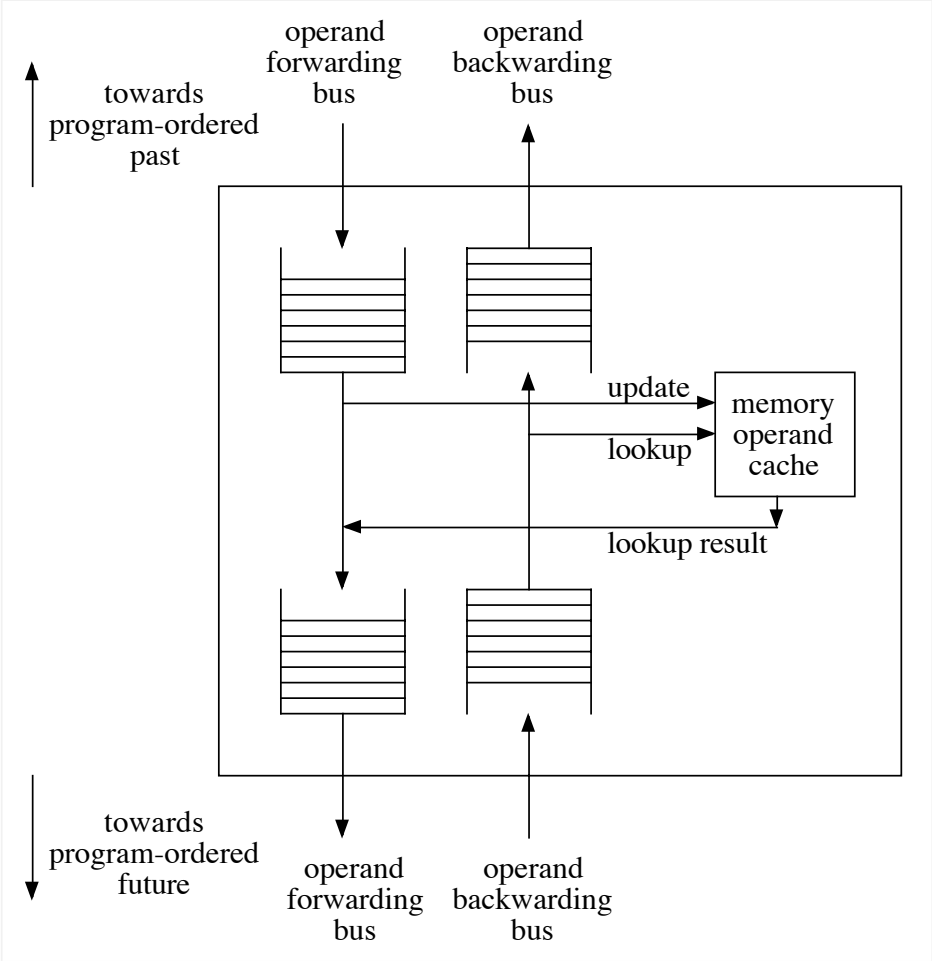

Figure 8.5: *An operand Memory Filtering Unit for OpTiFlow.* Shown is an operand filter unit for memory operands. This is also often just called a Memory Filtering Unit. It serves to filter out unnecessary operand forwards and backwards to save bus bandwidth.

to the backwarding request was not found in the cache, then the backwarding request is queued to the output FIFO for the output operand backwarding bus.

The organization for the cache is a design choice and can take on firms similar to regular memory caches. Specifically, fully associative, direct mapped, or set associative caches are possible candidates. The use of the cache can reduce the number of both operand forwards and operand request backwards that need to pass through the operand forwarding unit (both directions respectively). We term this new type of operand forwarding unit an *operand filtering unit*. This term reflects the fact that certain operand transfers are filtered out of being passed along when the corresponding operand is already present within the cache and meets certain criteria. Since the operand filtering unit we have so far described is used for memory type operands, it is sometimes also referred to as a *memory operand filtering unit* or more simply a *memory filtering unit*. The idea of an operand filtering unit can also be applied to register and predicate operands and these are described next.

8.3.2 Register operand filter

Just as memory operands could be filtered using the type of forwarding unit described previously (using an operand cache of some sort), so too can register type operands be filtered. Although an operand cache can be used for register operands (the register address would form part of the indexed lookup), a further optimization is possible due to the very small address space of all machine registers. Most machine architectures only have on the order of 64 architected registers or less. For our present purposes, we only need to consider the address space size of the architected registers and not all physical registers that may actually be present within the microarchitecture. We are already effectively performing a full renaming of registers through the use of the time-tag, so only the size of the architected register space need be considered here. Since the number of registers is relatively small compared with memory locations, it is possible to replace the operand cache type storage structure with a simpler array of operands. For example, if the machine architecture contained 64 architected registers, the array of operands would only have to have 64 entries. This is the simplest case for when no Disjoint Eager Execution (DEE) is performed (see Chapter 2 for a brief introduction to DEE computing). In the case when DEE computing is also performed, the number of array entries needed would be the product of the architected registers of the machine architecture and the number of total paths the machine is designed to handle simultaneously (mainline paths plus all DEE paths). This type of filtering unit is shown in Figure 8.6.

Similar to the memory filtering unit previously described, this type of unit is termed a *register operand filtering unit* or just *register filtering unit* for short. The operation of this filtering unit is similar to that of the memory filtering unit except that all stored operands are only register operands. Forwarded register operands (arriving from program-ordered past) update the

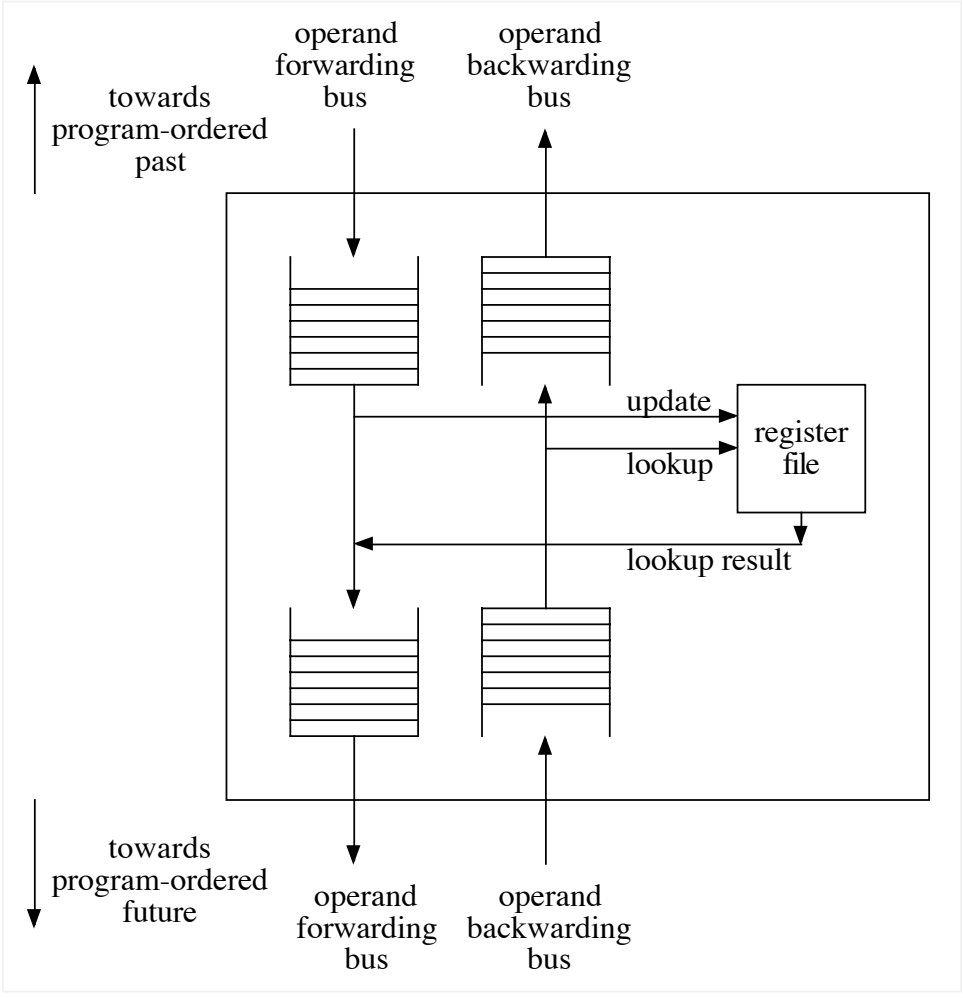

Figure 8.6: *An operand Register Filtering Unit for OpTiFlow.* Shown is an operand filter unit for registers. This is also often just called a Register Filtering Unit. It serves to filter out unnecessary operand forwards and backwards to save bus bandwidth.

corresponding register operand in the operand array. If the value of the register operand has not changed, it need not be further forwarded, otherwise it is further forwarded. Backwarding requests are also handled similarly to those of the memory filtering unit. When a backwarding request arrives (from program-ordered future), a lookup is made to the operand array. If the corresponding register operand is present (has not been invalidated for any reason), it is forwarded as a response to the request and the request need not be further backwarded. Else, if no valid corresponding operand is present in the array, the request is further backwarded where it will eventually be satisfied with a response. Backwarding requests are always eventually satisfied in the case of OpTiFlow from the architected register file if the request succeeds in getting backwarded to it before being satisfied by any intervening register filtering unit.

As in the case with the memory filtering unit, operand transfer bus bandwidth is saved when a valid operand exists within the filtering unit.

8.3.3 Predicate operand filter

In this section we address the issue of managing the forwarding and backwarding of predicate operands (with the possible filtering of them also). The predicate type of operand is somewhat less intuitive than either register or memory operands so we will provide a little bit more detail about how they are handled by their associated filter unit. The idea of a microarchitectural predicate operand (an operand of a type not visible within the architecture of the machine) was first introduced in Chapter 7. In that chapter we showed how the dynamically executing instruction stream of a program can be predicated. In that scheme, each instruction is individually predicated within the microarchitecture in a way that is entirely hidden from the architecture (and more specifically the instruction set architecture) of the machine itself. Even an instruction set architecture that contains predicate registers and predicated instructions can be dynamically predicated within the hardware of the machine since any architectural predication just appears to be additional register data flow to the underlying hardware. The case of handling predicate operands is somewhat more complicated than that of either memory or register operands but it will still share the basic hardware structure of the other filtering units.

In Chapter 7, microarchitectural operands were treated in a way somewhat similar to register operands. Specifically, the microarchitectural predicate operands were forwarded to subsequent instructions in a similar way as register operands were. These operands were tagged with time-tags just as the register operands were and were snooped and snarfed in a somewhat similar way. Now with the introduction of register and memory operand forwarding, it should be possible to also forward microarchitectural predicate operands similarly. We will do this using a new type of operand forwarding unit that we will term a *predicate forwarding unit*.

As was introduced in Chapter 7, predicate operands are of two types. These two types are: *fall-through predicates* and *target predicates*. The predicate operand type is a little bit more abstract than either the register or memory operands as they are not a single item quantity as those other operand types are. Rather, the idea of the predicate operand as it exists while being conveyed between instructions (ISes) really consists of the associated operand transfer types that are used to transmit it around the execution window. But the management of the predicate operand from one instruction to another is still remarkably similar to the register and memory types. As was the case with register and memory operands, a simple operand forwarding-backwarding unit arrangement as depicted in Figure 8.4 can be used to forward predicate operands. Although the simple repeater unit is adequate, some degree of predicate filter is also possible and is now introduced.

Figure 8.7 shows our operand predicate forwarding-backwarding unit that also contains some capability to filter out some unnecessary operand transfers. As the Figure shows, many of the same elements of the other operand forwarding-backwarding units are present in the operand predicate unit. Specifically, FIFOs are present to decouple the activity on the related buses from the predicate register and table lookups that occur within the unit. Unlike the register or memory filter units, the stored state for this filter unit consists of two types. These are the: single fall-through predicate, and a table of branch target predicates. The target predicate table is of a fixed size but is otherwise sized according to a desired design point. Further the target predicate table is essentially maintained as a cache of target predicates. Hits in this cache serve to allow for filtering of transfers, while misses require table updates.

Each of the two predicate types (fall-through and target) actually consist of a set of sub-state. For the fall-through predicate, this state consists of:

- a valid bit (Vf)
- a fall-through time-tag (TTf)
- the one bit value of the fall-through predicate (Pf)

The valid bit (Vf) is used to indicate whether the (so called) fall-through predicate is even valid or not. It is initialized as FALSE when a new predicate filter unit is assigned to participate in the operand transfer flow within the execution window. The fall-through time-tag (TTf) is used to hold the last time-tag value that was snarfed for the fall-through predicate. The last state is the one bit value of the fall-through predicate itself (Pf). For the target type predicate (of which there may be several depending on the size of the target predicate table), its state consists of:

- a valid bit (Vt)
- a target instruction address (IA)

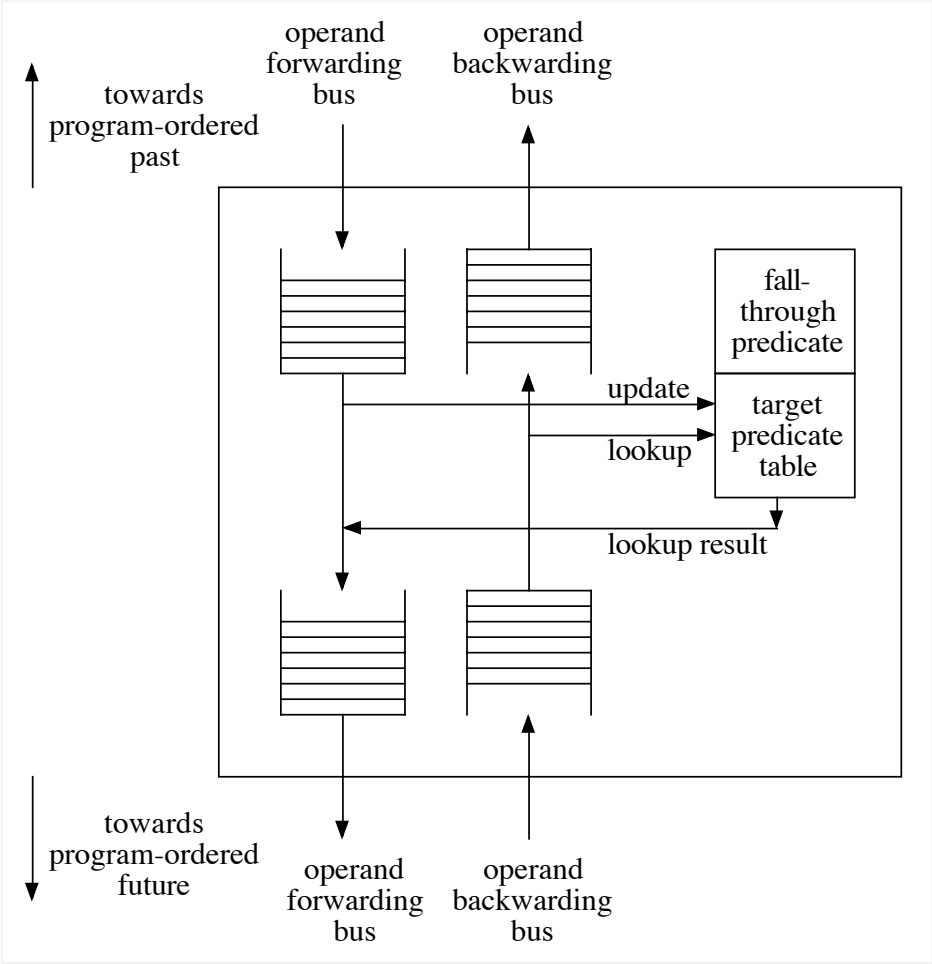

Figure 8.7: *An operand Predicate Filtering Unit.* Shown is an operand filter unit for handling microarchitectural predicates. This is often simply termed a Predicate Filtering Unit. It serves to forward and filter out unnecessary operand forwards and backwards to save bus bandwidth. Unlike other operand types, predicate operands are of two types: a fall-through predicate, and one or more target predicates.

- a target predicate time-tag (TTt)

Like the corresponding target predicates that are maintained within ISes, the existence of a valid table entry itself indicates a TRUE valid for the associated target predicate (addressed by the time-tag value). The absence of a target predicate entry (like at lookup time) indicates an unknown value for the associated target predicate. When the table is searched (done associatively), matches only occur for entries that have their valid bits set. Snoop comparisons are done using the time-tag values of the incoming operand transfer and the existing table entry. Like the operand table in the memory filter unit, the target predicate table could also be of a variety of organizations that are analogous to common memory caches. The target predicate table described so far corresponds to a direct mapped cache for memory use. Obviously, set associative organizations (indexed and tagged on parts of the entry time-tags) are also possible. How the various predicate state described so far is updated is discussed in relation to the operand transfers that come into the unit. This is discussed next.

The predicate operand filter unit accommodates both forwarding and backwarding as there exists both forwarding and backwarding predicate operand transfers. There are three forwarding operand transfers and one backwarding operand transfer (all previously discussed in Chapter 7). The three forwarding operand transfers are:

- fall-through predicate
- target predicate
- target predicate invalidation

The one backwarding operand transfer is:

- resend predicates

Maintenance of the predicate information within the unit occurs very similarly to what occurs within an IS, except that what might have been non-matching snoops for an IS have to be accommodated for the corresponding maintenance within this unit. The logical operations that are performed on the reception of the various predicate operand transfers are described in the subsequent subsections. The filtering out of predicate operand transfers is also indicated when it is possible.

reception of a Fall-through Predicate operand transfer

This operand transfer arrives to the unit from program ordered path on the incoming operand forwarding bus. A check is made with the existing fall-through predicate (of which there is only one) if one is available. An existing fall-through predicate is present if the valid bit (Vf) for

that predicate is currently TRUE. If no existing fall-through predicate is present, the fall-through predicate state is set with information from the incoming operand transfer. The time-tag of the fall-through predicate state (TTf) is set from the time-tag field of the transfer and the valid bit is set TRUE. Further, the incoming predicate operand transfer is queued to be farther forwarded onto the output FIFO and onto the output operand forwarding bus. No filtering is achieved in this circumstance. If an existing entry was present (the valid bit was TRUE) then a check is made to see if the time-tag of the incoming transfer is equal to or greater than the existing time-tag associated with the fall-through predicate. If the incoming time-tag is less than the existing time-tag (TTf), the incoming transfer is dropped (and a filter has occurred). If the incoming time-tag is greater, the existing time-tag is updated and the transfer is queued to to the output forwarding bus (no filter occurs). If the incoming time-tag is equal to the existing time-tag value, a further check is made to see if the value of the incoming predicate is different from the existing one (Pf). If the values are the same, the incoming transfer is dropped and a filter of the operand has occurred.

reception of a Target Predicate operand transfer

Upon receipt of this type of operand transfer two different operations occur in parallel. The first is the same processing that occurs for the receipt of a fall-through predicate operand transfer that was described in the previous subsection. The second operation consists of a snoop check of the target predicate table. Matching occurs with a comparison of the incoming and existing time-tag values. If no existing entry is present, a new entry is allocated (by evicting an existing entry if need be) and the operand transfer is queued for output (into future program ordered time). No filter of the operand has occurred in this circumstance. If an entry is present (its valid bit is set) that matches the incoming entry, then the incoming transfer, a further check is made to see if the incoming instruction address matches that of the entry in the table. If a match occurs the incoming operand transfer is dropped. and a filter has occurred. If a match does not occur, then the existing entry is updated with the new instruction address and the operand transfer is queued for output into program ordered future (no filter occurs). The check for the instruction address is required in the case when the originating control-flow chance instruction also changed its target address that it wants control-flow to check to. Although this will not occur with traditional conditional branch instructions, it does occur for indirect jumps (computed GOTOs).

reception of an Invalidation Target Predicate operand transfer

Upon receipt of this operand transfer (from program ordered path) a snoop check is made on the target predicate table. Matches occur with the comparison of the incoming and existing time-tag values. If no entry is found, the incoming invalidation target predicate is queued for forwarding. If

an entry is found, that entry is invalidated (its valid bit set to FALSE) and it is also queued for forwarding. No filtering occurs upon receipt of this type of operand transfer.

reception of a Resend Predicates operand transfer

This type of operand transfer arrives from the program ordered future on the incoming backwarding bus interface. Upon recept it is simply queued for further backwarding on the outgoing backwarding bus. Generally no filtering can occur with this type of operand transfer. The reason is that the target predicate table located within the predicate filtering unit does not (in general) contain all of the possible target predicates that there may be inside the execution window of the machine. In order for all of the possible target predicates within the execution window to be accommodated, the size of the target predicate table would have to be equal to the size of the execution window itself. If this latter condition occurs (through a machine design intent), then filtering can occur since the target predicate table could be looked (associatively) by the instruction address of the requesting IS and all matching target predicates can be immediately forwarded (filtering out an otherwise necessary backwarding request).

8.3.4 Combining operand filtering units

Although we have described each of the various operand filter units (register, memory, and predicate) as separate units each with their own incoming and outgoing incident buses, an actual implementation can combine these into a single logic block if desired. In this latter case, all buses could be shared rather than being different and designated for the different types of operand transfers they handle. The operational logic of each type of unit and its associated state would remain the same, but the interfaces to the operand transfer fabric would be shared. This is usually the most desirable design decision as any remaining buses are used most efficiently without operand-type fragmentation waste (to borrow an idea from another venue) occurring. This is to say that when all forwarding buses of any type and separately all backwarding of any type are combined, there is no inefficiently possible with the bus utilization if there is not enough operand transfers of one type to fill up its own designated bus fabric. This amounts to a sort of statistical multiplexing of buses based on operand type and it is almost always the best design decision. Of course, a variety of operand interconnection fabrics is possible and that is a topic beyond the scope of the present work. But if it ever was the case that certain interconnection fabrics were employed and were also highly optimized for certain operand transfers of a certain type (the type here being: register, memory, or predicate) then the various incoming and outgoing operand transfer interfaces on the various filter units could indeed remain separated.

8.3.5 A physically scalable microarchitecture

With the introduction of the various forwarding-filter units that we have discussed, we are now in a position to introduce a new class of computer microarchitecture that can physically scale. Using the forwarding-filter units to rather arbitrarily break up long bus paths, new possibilities become available for dramatically increasing the number of physical components that can be employed to work towards executing a single instruction stream. This is the most difficult type of instruction level parallelism to achieve and the introduction of the idea of forwarding-filter units for operands is a key component in this evolution.

8.4 Summary

We have introduced a way to extend the operand transfer buses of Resource Flow microarchitectures through the construction and addition of an operand forwarding unit. This unit allows for the logical segmentation of operand transfer buses into smaller lengths. This segmentation solves the problems associated with long wires and propagation delays that would be present with long bus lengths. Through the breaking up the operand transfer buses into fixed length segments, this allows for the physical scalability of a Resource Flow microarchitecture. Larger numbers of machine components, such as issue stations and execution function units can be included in a physically scaled machine than could be otherwise. The expectation is that this can possibly produce execution performance improvements through extraction of larger amounts of instruction level parallelism from the microarchitecture.

In addition to the introduction of a minimal operand forwarding unit, we have also introduced the idea of enhancing forwarding units to also perform an operand filtering function. Operand filtering units filter out (do not forward or backward) operand transfers that can be satisfied locally within the filtering unit itself. Different types of filtering units are used for the different types of operands: memory, register, and predicate. The different operand filtering units may have their own buses for operand transfer or they may share a common set of buses. With the filtering out of unnecessary forwards and backwards of operand transfers, bandwidth on the operand transfer buses is freed up for higher local bus segment utilization than otherwise possible. This filtering benefit allows for less overall bus bandwidth to be supplied with physical buses for a given machine size (in components) than otherwise would be needed.

Chapter 9

Register and memory access interval characterization

In the previous chapter we introduced the idea of operand forwarding units, along with their enhanced variety generally termed operand filter units. Forwarding units of three types were introduced where each corresponded to a basic type of operand that they handle. The three operands types were: register, memory, and predicate. The basic goal of an operand forwarding unit was to electrically segment buses carrying operands from one set of issue stations (IS), which were introduced with the Resource Flow execution model (see Chapter 3), to others that are logically positioned in future program ordered time as compared with the originating stations. Concomitantly, these units also backwarded requests for operands from from younger (in program ordered time) issue stations to older ones. Achieving this basic goal allows for the physical scalability of a microarchitecture using the Resource Flow execution model. Further, the idea of forwarding units was also extended to perform an operand filtering function. These latter units were thus termed operand filter units. These filtering units provided the additional function and operational advantage of reducing the number of operands that need to be forwarded, thus freeing up operand bus transport bandwidth and congestion. However, the use of any type of forwarding unit introduces a delay in the transmission of operands from the input receiving bus to the output sending bus. Operand buses cannot be arbitrarily long so the use of these units is required, but the spacing of the units (in terms of the number of issue stations allowed on each physical bus segment) is a design parameter. This parameter has been termed a bus *span*. In an aid to helping the designer in choosing possible bus span design points we would like to characterize the set of programs that we especially want to target in order to determine the suitability of the whole bus span and forwarding unit idea given the targeted program behavior in terms of its operand usage. The operand usage that is important for our investigation is that of the intervals (in numbers of dynamic instructions,

and thus issue stations) separating operand writes to subsequent operand accesses (either reads or writes).

This chapter explores the access intervals (as measured in dynamic instructions executed) between assignment to registers (writes) and their corresponding `use`s (reads), as well as the assignment to memory locations (denoted by a memory address) and their corresponding `use`s. We define three types of access intervals that we want to explore. These access intervals are defined similarly for both register and memory operations. The exploration of these access intervals will serve to validate our machine design approach taken in Chapter 8) for achieving machine scalability within the context of Resource Flow microarchitectures. The significance of these access intervals is discussed in more detail later.

We have gathered data associated with each of our three access intervals, for both register and memory operations, through the execution of general purpose sequential program codes. This work was substantially inspired by the prior work of Franklin et al. [33] in their investigation of register traffic for use in the context of the Multiscalar-like microarchitectures. Prior work on memory locality has been substantial including such work as Madison et al. [90], Verkamo et al. [159], and more recently that of Phalke et al. [160] with their Inter-Reference Gap model. Eeckhout and Bosschere have also investigated access intervals for registers (they didn't consider memory) [27] but their goal was different than ours. They attempted to distill the access interval characteristics of registers into work-load parameters that could be used to drive an hybrid analytical-statistical modeling technique for new microarchitectures.

The present work builds on and extends that of Franklin et al. and also to some extent on that of Eeckhout and Bosschere by providing additional information on register operand traffic that was not previously reported. We also extend the previous work of both Franklin et al., Eeckhout et al. and Phalke et al. by applying our same measurements for register traffic operations to memory traffic operations.

The rest of this chapter is organized as follows. Section 2 gives some motivation for why we wanted to perform access interval characterization. Section 3 presents our definitions of the access intervals that we studied along with some significance of those intervals for microarchitectural design decisions. Section 4 presents our characterization results. The results are presented in two parts: 1) for the register operations and 2) for memory operations. We summarize the present work in Section 5.

9.1 Definitions and significance of intervals

There are several possible intervals that can be defined with regard to writes to a variable (whether a register or a memory location) and the corresponding reads of the same variable. We only

explore three of these in this present work since other possible intervals do not have as important significance as the three types of intervals that we have selected. We first provide some definitions for the intervals that we have explored and then give some possible significance to those intervals for microarchitectural design purposes.

9.1.1 Definitions of intervals

We define all intervals in terms of the definition of a *variable instance* (hereafter referred to as a *def*) and the corresponding subsequent *use* of the same instance of the variable. A write to any variable always constitutes a `def` of a new variable instance. A read to a variable constitutes a use of the associated variable instance. For our purposes, reads and writes occur on variables while `def`s and `use`s occur for variable instances. Variable instances are uniquely identified by both their variable address and their associated `def` (as identified in dynamic program ordered time). For registers, the address is generally just its name (generally its architected ordinal index to the architected register files). For memory variables, the address is that of the memory location itself (architected memory address). As alluded to already, a write to a variable destroys the previous instance of the variable while simultaneously creating a new instance of that variable. This idea of variable instance is the same as that of Franklin et al. [33]. A write to a variable is also assumed to constitute a `def` of a new variable instance even if the value assigned to the variable is the same as that which it had already. Access `use`s of variable instances occur when the associated variable is read for any architected reason, whether this is explicit or implicit to the execution of an instruction.

We explore the following three types of access intervals on both registers and memory locations:

- *access-use*
- *useful-lifetime* (or *def-last-use*)
- *def-use*

The *access-use* interval is defined as the number of dynamic instructions from a read of a variable to the closest of a preceding read or write to the same variable. An access-use interval is therefore a property of a read of a variable. Note that there may be many `use`s of the same instance of a variable. Each read of a variable will therefore usually have a different access-use interval associated with it since the number of dynamic instructions from the `def` of the variable instance to the current use is usually different. Two `use`s may have the same access-use interval when, for example, each is associated with an input operand to a single instruction. Our definition for the access-use interval is the same as the *inter-reference gap* from the work of Phalke et al. [160]. Franklin et al. [33] did not consider this type of access interval in their work.

The *useful-lifetime* is defined as the number of dynamic instructions between the write of a variable and the **last** read of the same variable before the subsequent write to the same variable.

This interval is also sometimes referred to as the *def-last-use* interval since only the last use of the variable instance is taken into account when determining this interval. This interval is therefore a property of the write to a variable. Each write of a variable therefore only has one useful-lifetime interval associated with it. If there are, for example, three `use`s following the `def` of a variable instance, only the last such use is used to determine the useful-lifetime of the variable instance. Note that the use-lifetime of a variable instance can have a value of zero. The term *useful-lifetime* is taken from Franklin [33]. This term is generally synonymous with the *liveness* of a variable instance [45]. It should be noted that the term *lifetime* is often confused with that of useful-lifetime. For our purposes, we define the term lifetime to refer to the interval from a `def` of a variable instance (write to a variable) to the subsequent write of the same variable (thus creating a new variable instance). This is generally different than the def-last-use interval for the same variable instance. This definition of lifetime is the same as that used previously by Franklin et al. [33]. We do not further explore lifetime intervals in this present work.

Finally, the *def-use* interval is the number of dynamic instructions from the read of a variable to the preceding write of the same variable. As the case was with the access-use interval, the def-use interval is a property of a read of a variable and each such read will generally have a different def-use interval associated with it. The exception occurs when two reads of the same variable occur in the same instruction (the same as was the case with the access-use interval above). Our definition for the def-use interval is also referred to as the variable instance *age* in the work by Franklin et al. [33] and Eeckhout et al. [27]. The idea of age is apparent since each use of a variable instance can be thought of occurring at a certain age of the instance as measured by the dynamic number of instructions since the associated definition of the same instance.

A simple, and quite contrived, code example to illustrate the meaning of the three intervals that we have defined is shown in Table Table 9.1. In this simple code example, the term c is an arbitrary immediate constant encoded within the instruction which is also generally different for each instruction. All of the instructions produce `def`s of a variable instance associated with their destination registers. Instructions I3 and I4 also produce `use`s of the registers *r2* and *r1* respectively. The def of register *r1* in instruction I3 allows for the determination of the useful-lifetime for the previous variable instance (from instruction I1) held in that register. In the present example, the useful-lifetime for that variable instance is calculated as 0. This result is due to the fact there there were no intervening `use`s of the register between instructions I1 and I3. Likewise, register *r2* is `def`ed in instruction I4 and this allows for the determination of the useful-lifetime for the previous variable instance held in that register (from I2). That is calculated as being being 1 since there was a use of that previous variable instance in instruction I3. Note that useful-lifetimes for a variable instance cannot be determined until a subsequent write of the corresponding variable is encountered. Similarly, both instructions I3 and I4 contribute a def-use interval with the value

Table 9.1: *Simple code example illustrating the different types of access intervals.* These machine parameters are used for all simulations.

label	instruction	event	intervals determined
I1	`r1 <= c`	def(r1)	
I2	`r2 <= c`	def(r2)	
I3	`r1 <= r2 + c`	def(r1), use(r2) useful-lifetime(r1)=0	access-use(r2)=1, def-use(r2)=1,
I4	`r2 <= r1 + c`	def(r2), use(r1) useful-lifetime(r2)=1	access-use(r1)=1, def-use(r1)=1,
I5	`r3 <= r1 + r2`	def(r3), use(r1), use(r2)	access-use(r1)=1, access-use(r2)=1, def-use(r1)=2, def-use(r2)=1

1 since each represents a use of the corresponding variable instance just one instruction after its associated def. Finally from instruction I5, since there are `use`s of both registers *r1* and *r2*, an access-use and def-use interval can be determined for each of these. Note that for register *r1* an access-use interval with value 1 is determined while a def-use interval of value 2 is determined. For register *r2*, both its access-use and def-use intervals have value 1 since there was a def of that register in just the previous instruction.

Note that an access-use interval and a def-use interval is always determined for each use of a variable instance, but that a useful-lifetime interval can only be determined on a def of a new variable instance. Although this example illustrated the determination of the three types of access intervals on registers, these are determined similarly for memory references.

9.1.2 Microarchitectural significance of the intervals

Each type of variable access interval that we are exploring may have different significance or consequence for making distributed microarchitectural design decisions. Although all of the intervals discussed previously share some similarity in their attributes, they tend to answer different microarchitectural questions. Firstly some types of intervals are associated with reads while others are associated with writes. This distinction is used when considering the applicability of each type of interval metric.

The useful-lifetime interval is a property of a write so data on these intervals would be useful when exploring the desired microarchitectural consequences of a write occurring. If, for example, the resulting operand from a write operation could be buffered locally near the execution units in the processor, on average it would only have to remain buffered until a number of following instructions

(in dynamic program order) equal to its useful-lifetime had an opportunity to snoop the buffer for the operand. After the appropriate number of subsequent instructions had an opportunity to snoop the buffer, it could be assumed that the likelihood of a further `use` of that operand is minimal and therefore the operand could be released or evicted from the buffer.

In contrast, since both the access-use interval and the def-use interval is a property of a read, microarchitectural decisions about how to best satisfy read operands might use the data associated with either of these types of access intervals. If we first consider the case of no intervening buffering or caching of operands between an operand sourcing instruction and its associated sink instructions, then the def-use interval data would be appropriate to use for possible design decisions. This is so because operand sinking type instructions could only be satisfied by either a previous operand sourcing instruction (allowing for operand bypass) or lacking that, the operand would have to be fetched from the appropriate centralized resource (architected register or future file in the case of a register, and the L1 data cache or load-store queue in the case of a memory operand).

For those distributed microarchitectures that can employ some sort of operand buffering or caching spatially close to the execution units, the access-use interval is the more appropriate metric to determine design decisions. This is so because a cache of some sort can retain a copy of the desired operand even though the original generation (original write) of the operand may have occurred in the distant past in the instruction stream. Either the cache could maintain the operand for a reasonably long period of time from a preceding write, or perhaps more likely, intervening reads keep the cache from evicting the operand needed by subsequent sink instructions. Further, a previous read could have resulted in the operand getting cached from the appropriate centralized resource without even a recent write of the operand having ever occurred.

9.1.3 Application for Resource Flow execution using filter units

For our present purposes (evaluating the merits of our design for physical scalability as outlined in Chapter 8) the useful-lifetime and access-use intervals are the appropriate choices for assessing the required or acceptable number of issue stations between successive placement of operand filter units (the bus span) before incurring excessive operand transfer delays. This is so due to the existence of buffering of the operands within the execution window with our Resource Flow execution model (Chapter 3) and considering the OpTiFlow microarchitecture (of Chapter 4) scaled using filter units (Chapter 8). In this context, intervals in terms of numbers of instructions correspond precisely with the distance between issue stations. For write operations (either register or memory), in additional to being transferred forward the write operand itself continues to reside within the originating IS during the time the instruction is in execution, being effectively buffered and available for access by subsequent reads. For register reads, the desired operand is also buffered some number of ISes

away (backward in program ordered time), either within an IS (one that is creating it through a register write) or within a preceding register filter unit. For memory reads, the desired operand may be in the memory hierarchy (at L1 or higher) or may be buffered the same as a register read is, being either in its originating IS or a preceding memory filter unit. With these considerations, the design question of how far a write operand needs to be transfered into future program ordered time (in units of IS) is most directly answered by examining the useful-lifetime interval program characterization. Likewise, for reads, the design question of how far back the operand request needs to be transfered (backward in program ordered time) is well answered by the access-use interval characterization data.

Minimizing total operand transfer delays without considering the workload access interval characteristics at all would indicate that the largest span possible be used (so that as few filter units are traversed as possible). However the largest span is also directly restricted by the electrical properties of the buses, drivers, and receivers, and as laid out in silicon. However, besides meeting the electrical requirements on bus spans, there is a reason for also minimizing bus spans when possible. Shorter bus spans allow for higher overall interconnection fabric transfer capacity than using longer spans. To clarify this, consider the extremes. If there was one operand transfer bus that served the whole of the execution window, then the total transfer bandwidth would be that of the single bus. However, if there were multiple buses serving the whole of the execution window, the total operand transfer bandwidth would be that of a single bus multiplied by the total number of buses. So some sort of happy balance on the length of bus spans is desirable. Evaluating the access interval data results can help serve to achieve this design optimization.

9.2 Benchmark statistics

In this section we present the results of accumulating our access interval data on ten benchmark programs. The programs are taken from the SpecINT-2000 benchmark suite. The following benchmark programs were used: BZIP2, CRAFTY, EON, GCC, GZIP, PARSER, PERLBMK, TWOLF, VORTEX, and VPR. All programs were compiled for the Alpha instruction set architecture (ISA). The specific compilation environment is given in Table 9.2. The simulations were carried out by first executing program instructions in a fast-forward manner up to a certain point in each benchmark and then switching into a functional simulation mode. The number of instructions fast-forwarded (skipped) for each benchmark program is termed that program's single-simulation SIMPOINT and was first described by Sherwood .et al. [123] The number of instructions fast-forwarded is different for each program. The block of instructions (100 million) following the SIMPOINT for each program are representative of the entire program behavior when only a single block of instructions can be executed. The SIMPOINTs for each program are given in Table 9.3 along with the specific

Table 9.2: *Compilation environment used for our benchmark programs.* The operating system, specific C-language compiler, and targeted processor (for the compilation) is given.

component	version
machine word size	64 bits
operating system	Digital UNIX V4.0F
C compiler	DEC C V5.9-008 on Digital UNIX V4.0 (Rev. 1229)
processor	21264

Spec2000 reference input used (where more than one was available). After fast-forwarding to the

Table 9.3: *Benchmark program SIMPOINTs and inputs.* Given here are the SIMPOINTs (in millions of instructions) used for each benchmark program and the Spec2000 reference input used when more than one was possible.

program	instructions skipped	input
BZIP2	45,900	program
CRAFTY	77,500	
EON	40,400	
GCC	39,000	166
GZIP	119,000	program
PARSER	114,700	
PERLBMK	1,200	make
TWOLF	106,700	
VORTEX	27,200	one
VPR	47,700	route

correct point in the respective benchmark programs, a period of functional simulated execution occurs where the functional hardware structures are warmed up. This period consists of one million instructions. Data on the program execution is accumulated for the subsequent 100 million instructions.

The next subsection presents the results for register access use. The following subsection presents the results for the memory access use. For both registers and memory, data is shown for access-use, def-last-use (useful lifetime), and def-use intervals.

9.2.1 Register access interval results

In this section, we show the register access interval results for our ten benchmark programs. For each interval type that we have explored, the results for the benchmarks are shown in two groups of five benchmarks each. Access (read) intervals are shown first, followed by useful-lifetime intervals, and finally the def-use intervals.

The data for register access-use intervals are shown in Figures 9.1 and 9.2. Results from benchmark programs BZIP2, CRAFTY, EON, GCC, and GZIP are shown in Figure 9.1 while the results from programs PARSER, PERLBMK, TWOLF, VORTEX, and VPR are shown in Figure 9.2.

The data for register def-last-use (or useful-lifetime) intervals are shown in Figures 9.3 and 9.4.

The data for register def-use intervals are shown in Figures 9.5 and 9.6. For better clarity, in Figure 9.7 we show the cumulative register access interval data over all of the benchmark programs. That figure shows, in order, all three of the access intervals that we explored: access-use, useful-lifetime, and def-use. The access interval results of all programs were added for each of the three access interval types respectively and then the densities and distributions were then calculated.

Before considering the more significant aspects of the data results, we want to note the meaning of zero length intervals for each of the types we have examined. These are present in the result data, but are not as mysterious as might seem at first. For the access-use intervals, it simply means that two or more read accesses occurred within the same instruction. If only a single read of a single architected register occurs within an instruction, then the intervals would be at least one in length (the distance to the previous instruction). Having two reads of the same architected register within a single instruction is not entirely uncommon and might, for example, indicate something like an ADD instruction uses the same architected register twice (to derive double its value). A zero length interval for the useful-lifetime interval type simply means that the variable instance didn't have any (useful lifetime that is)! This can also occur in code where, for example a simple single-sided Hammock conditional branch code structure exists. The compiler (or human) sets the value of a register before the conditional branch, and then sets it within the Hammock if the predicate of the conditional branch resolves to FALSE. This is not an uncommon case at all, but other cases can exist also in more complicated code. For the def-use interval type, a zero length value would indicate the same sorts of conditions just described for the useful-lifetime interval type; namely, no `use`s associated with the `def`. These same sorts of considerations hold for memory access intervals also.

Viewing the results across all benchmark programs (referencing the distribution graphs of Figure 9.7) some general observations can be made. For the access-use type intervals on average about 85% of all intervals are within 10 dynamic instructions and about 95% of all intervals are within 20

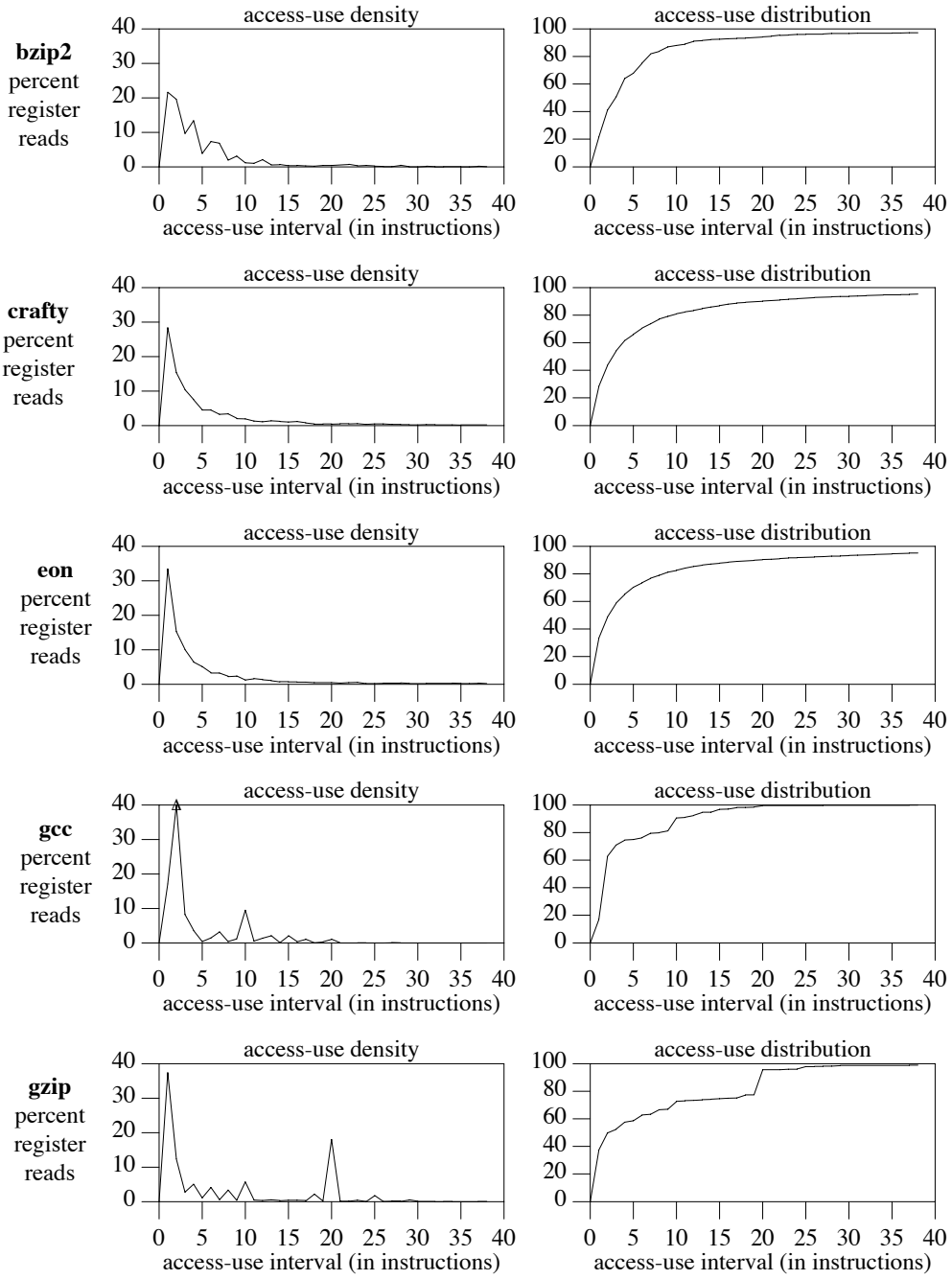

Figure 9.1: *Register Access-Use Intervals.* Data results for the BZIP2, CRAFTY, EON, GCC, and GZIP programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

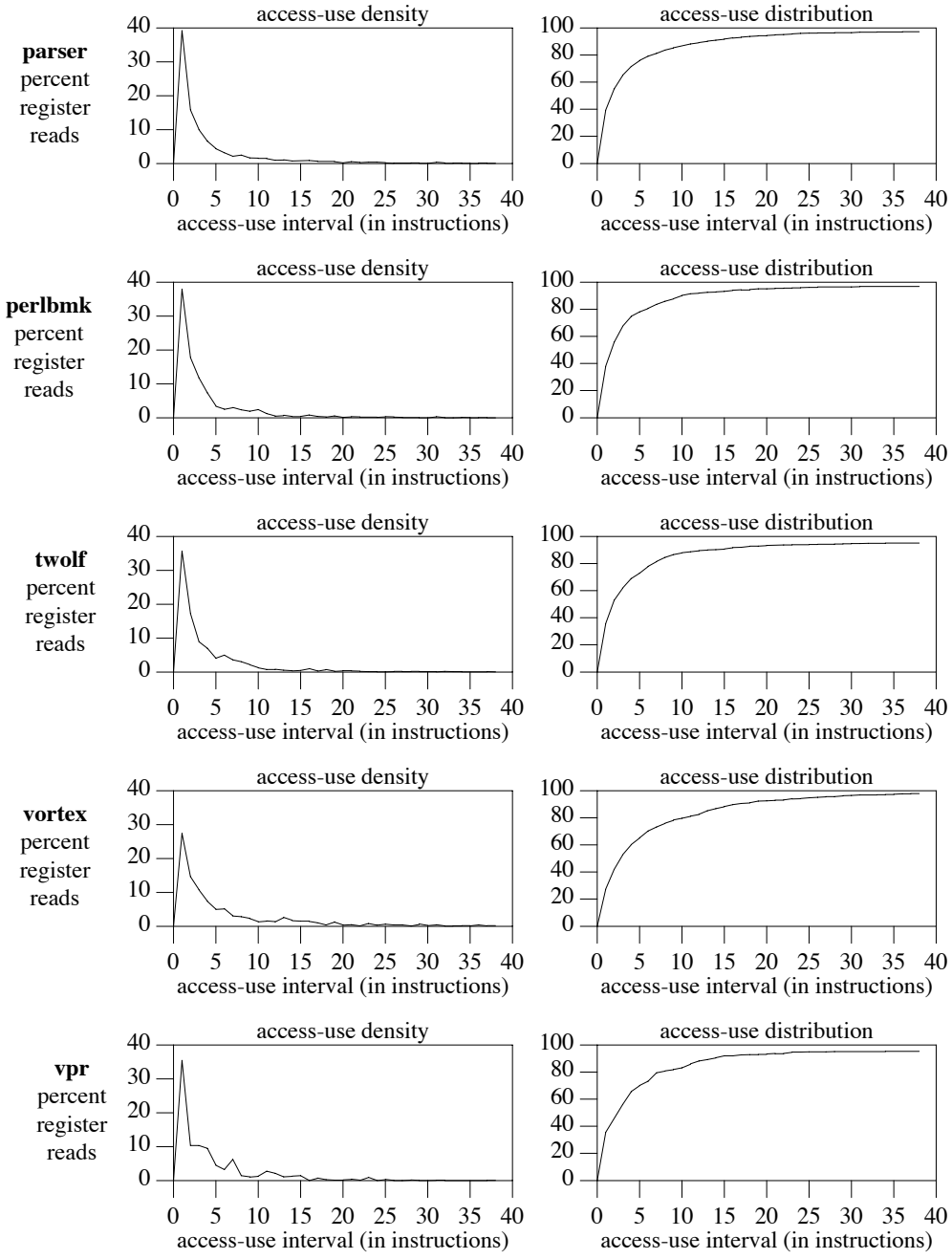

Figure 9.2: *Register Access-Use Intervals.* Data results for the PARSER, PERLBMK, TWOLF, VORTEX, and VPR programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

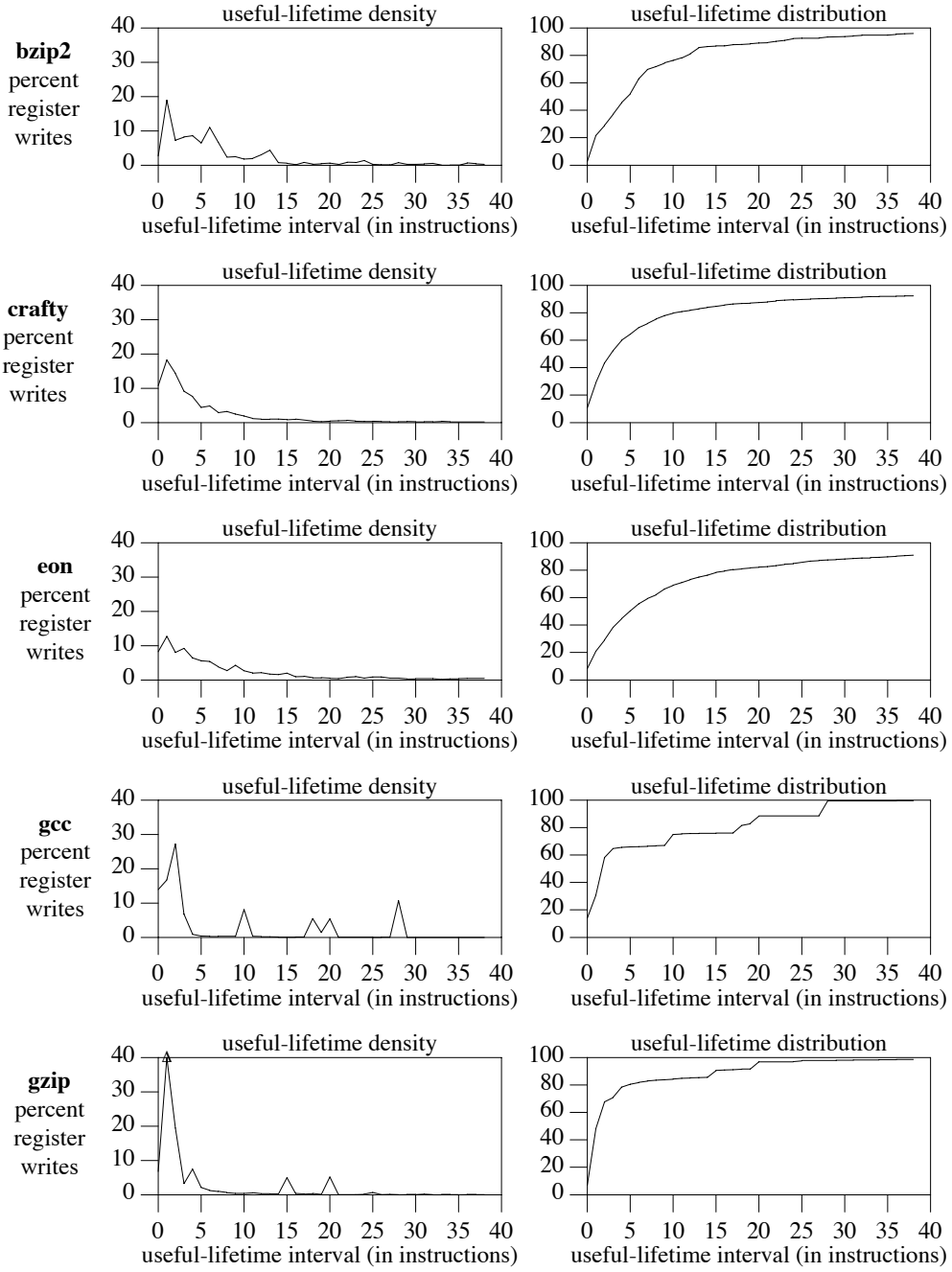

Figure 9.3: *Register Def-Last-Use Intervals.* Data results for the BZIP2, CRAFTY, EON, GCC, and GZIP programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

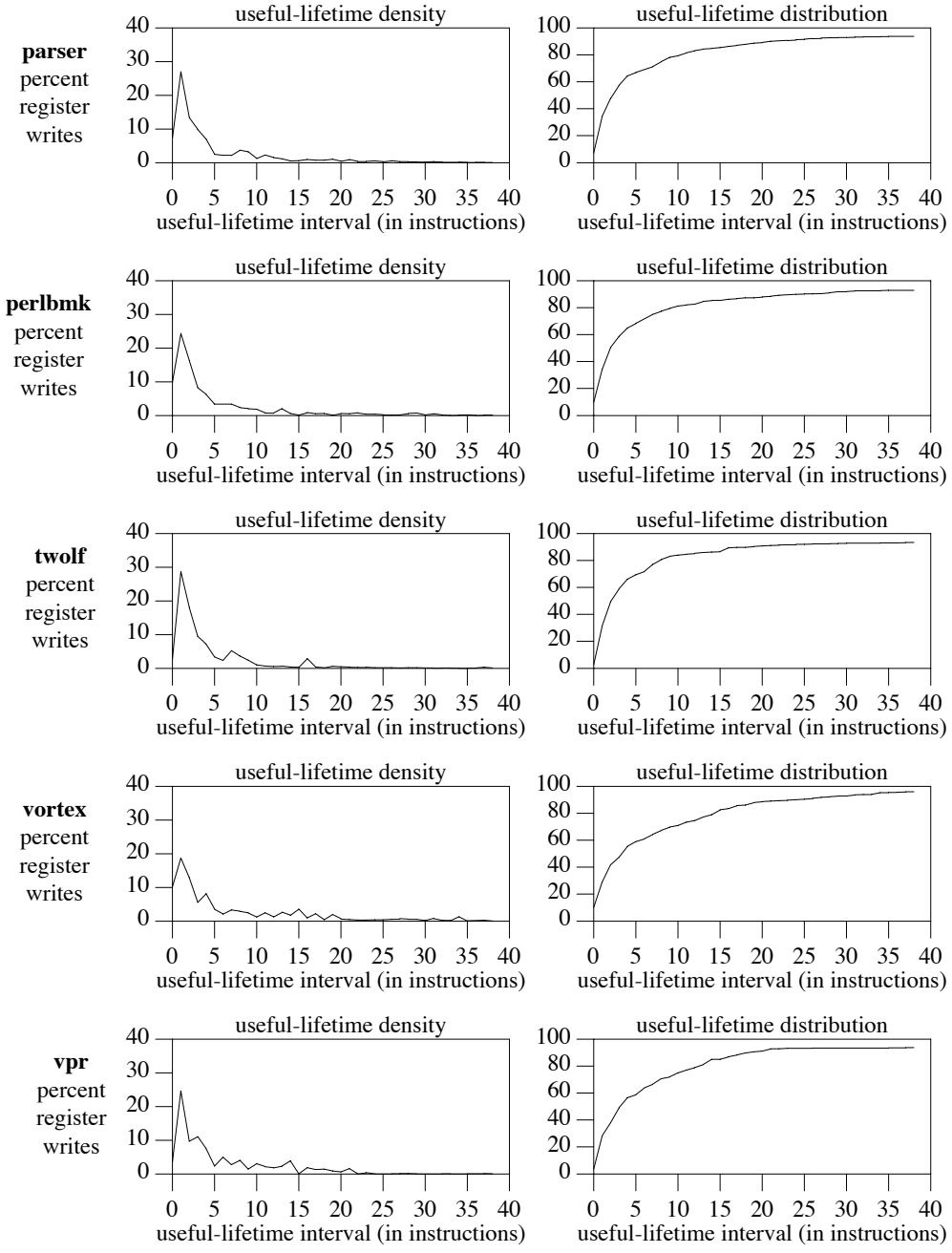

Figure 9.4: *Register Def-Last-Use Intervals.* Data results for the PARSER, PERLBMK, TWOLF, VORTEX, and VPR programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

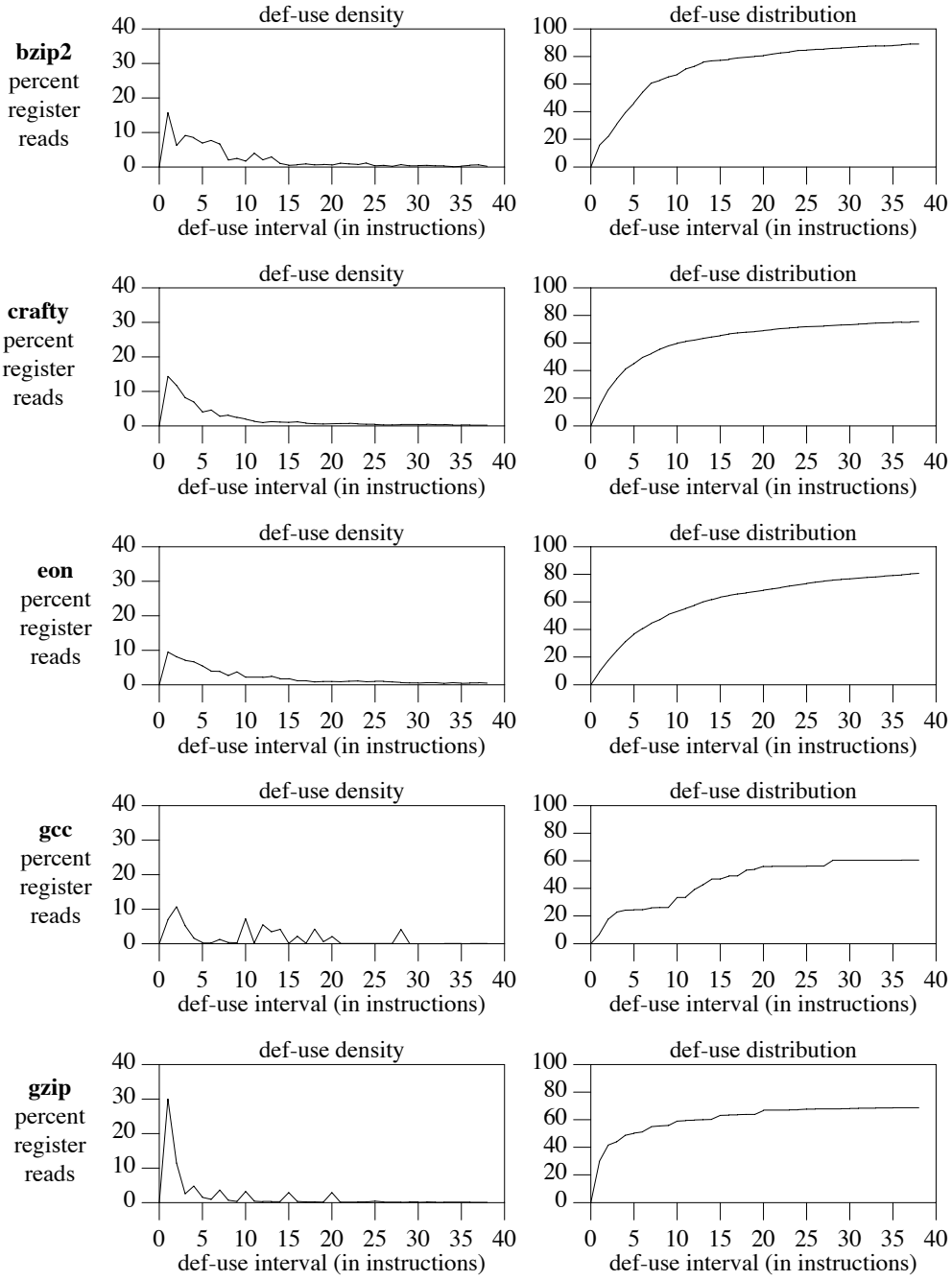

Figure 9.5: *Register Def-Use Intervals.* Data results for the BZIP2, CRAFTY, EON, GCC, and GZIP programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

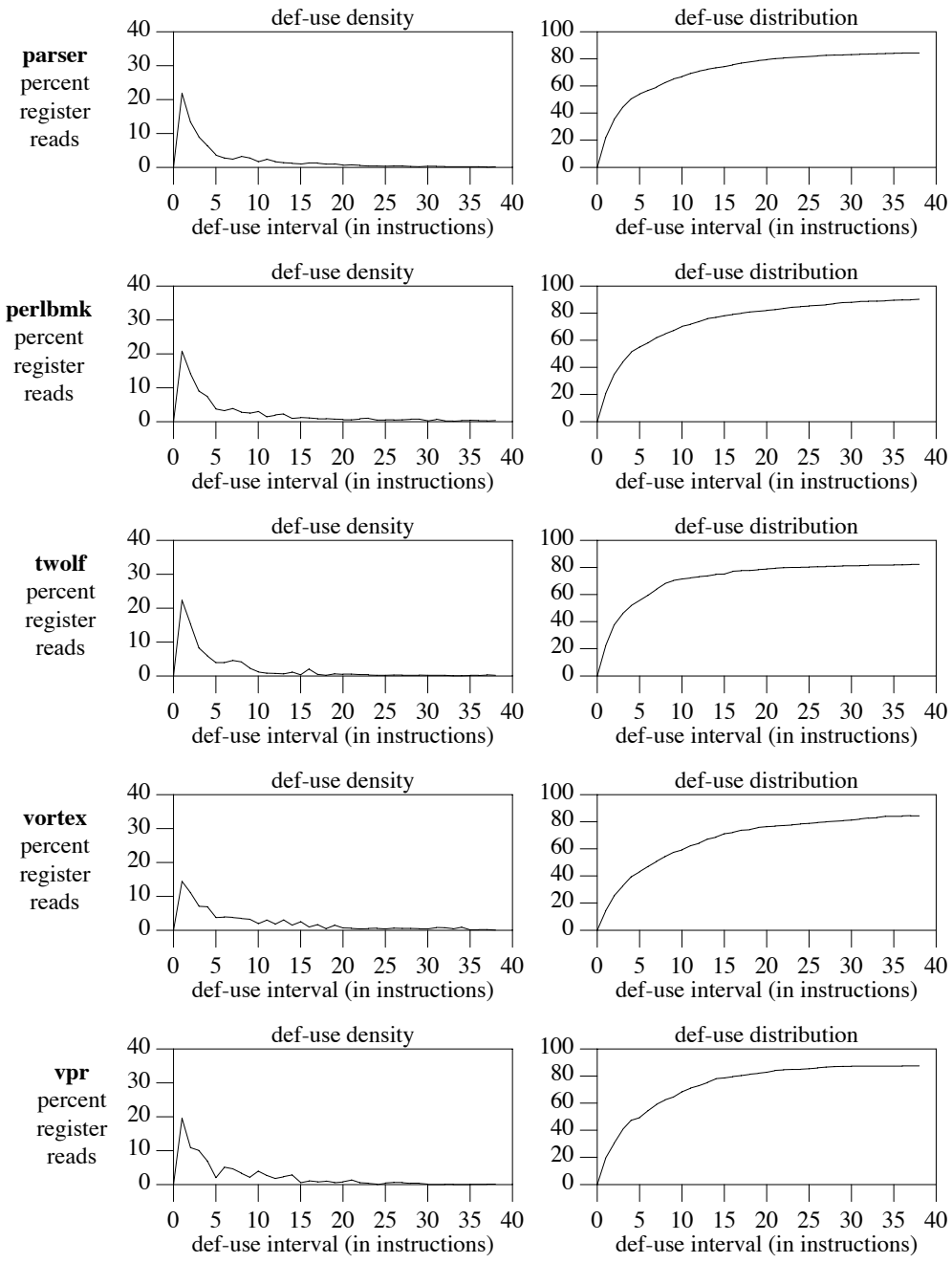

Figure 9.6: *Register Def-Use Intervals.* Data results for the PARSER, PERLBMK, TWOLF, VORTEX, and VPR programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

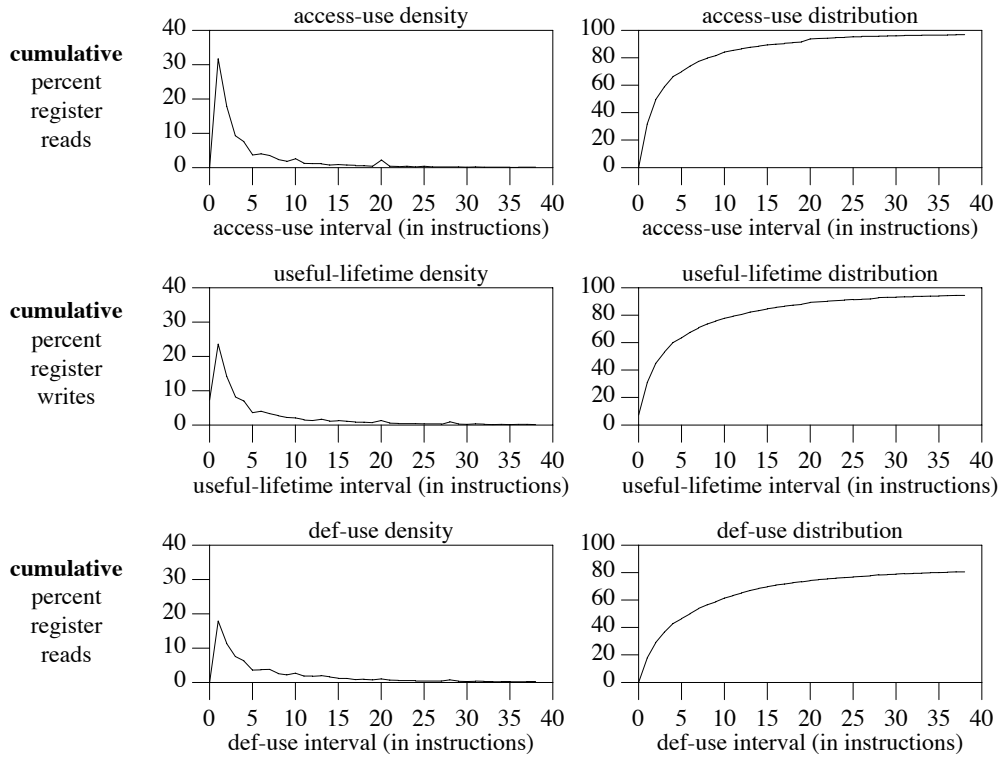

Figure 9.7: *Cumulative Register Intervals over all benchmarks.* The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

instructions. For the useful-lifetime intervals about 75% of all intervals are within 10 instructions and about 85% are within 20 instructions. Finally for the def-use intervals, about 60% of them are within 10 instructions, about 70% within 20 instructions.

These results indicate that if, for example, register operand filter units are placed at intervals of approximately 10 to 20 issue stations apart that most of the register operand transfer traffic, either register writes (a forwarding transfer) or register read requests (a backwarding transfer), would not even need to pass through a unit in order to be delivered to the next instruction needing that operand. For those operands that do need to pass through an operand storage unit, likely only one such unit needs to be traversed before delivering the operand to the instruction needing it. Since any operand storage unit incurs a delay in passing an operand through it, the smallest number of such units in the path of each operand is desirable.

9.2.2 Memory access interval results

In this section, we show the memory access interval results for the same set of benchmark programs. The data are presented in three sets. The first set of data is for register access-use intervals. The second set of data is for register useful-lifetimes, and the third is for register def-use intervals.

The data for memory access-use intervals are shown in Figures 9.8 and 9.9. Results from benchmark programs BZIP2, CRAFTY, EON, GCC, and GZIP are shown in Figure 9.8 while the results from programs PARSER, PERLBMK, TWOLF, VORTEX, and VPR are shown in Figure 9.9. The data for memory def-last-use (or useful-lifetime) intervals are shown in Figures 9.10 and 9.11. The data for register def-use intervals are shown in Figures 9.12 and 9.13. In Figure 9.14 we show the cumulative data over all benchmark programs. That figure shows all three of the access intervals that we explored: access-use, useful-lifetime, and def-use. The access interval results of all programs were added for each of the three types respectively and then the densities and distributions were then calculated. As can be seen from the various figures for the memory access intervals, unlike for the register intervals, these intervals are much longer on average and somewhat more varied from program to program than the register intervals were. The more prominent spikes in several of the density graphs is evidence of high frequency looping over memory locations. Also, it is interesting to note that many programs exhibit a significant number useful-lifetimes of zero or close to zero (particularly GCC). A zero-length useful lifetime represents a memory location that is later overwritten without an intervening read (an abandoned write). This can happen, for example, when writing to output file buffers (where the effective corresponding reads are being done by hardware DMA and are not within the CPU itself), This can also be due to control flow changes that abandon written variables. In the case of GCC, for the block of code being executed (supposedly representative of much of the program) it is looping on code that has some zero length memory variable useful lifetimes but with references to other variables inside the loop that gives rise to very large def-use intervals. The large def-use intervals are caused by defining a variable before entering the a high iteration loop and then referencing the same variable within the loop. The existence of zero length useful-lifetime intervals should be an indication to compiler writers to try to avoid writing to memory variables needlessly for those cases where the location may be overwritten later. They should have tried to use registers as temporaries instead. Similarly, the existence of large def-use intervals shows that a variable that is needed within a look (for a calculation perhaps, whatever) should always try to be placed within registers. Of course, when no additional registers are available for the compiler writer for holding loop invariants, memory locations have to suffice. This sort of situation may simply show that the operation being performed in the program may just be too involved and that a larger architected register set would have been advantageous.

Before returning to the data, it is helpful to recall how memory read operations can acquire their

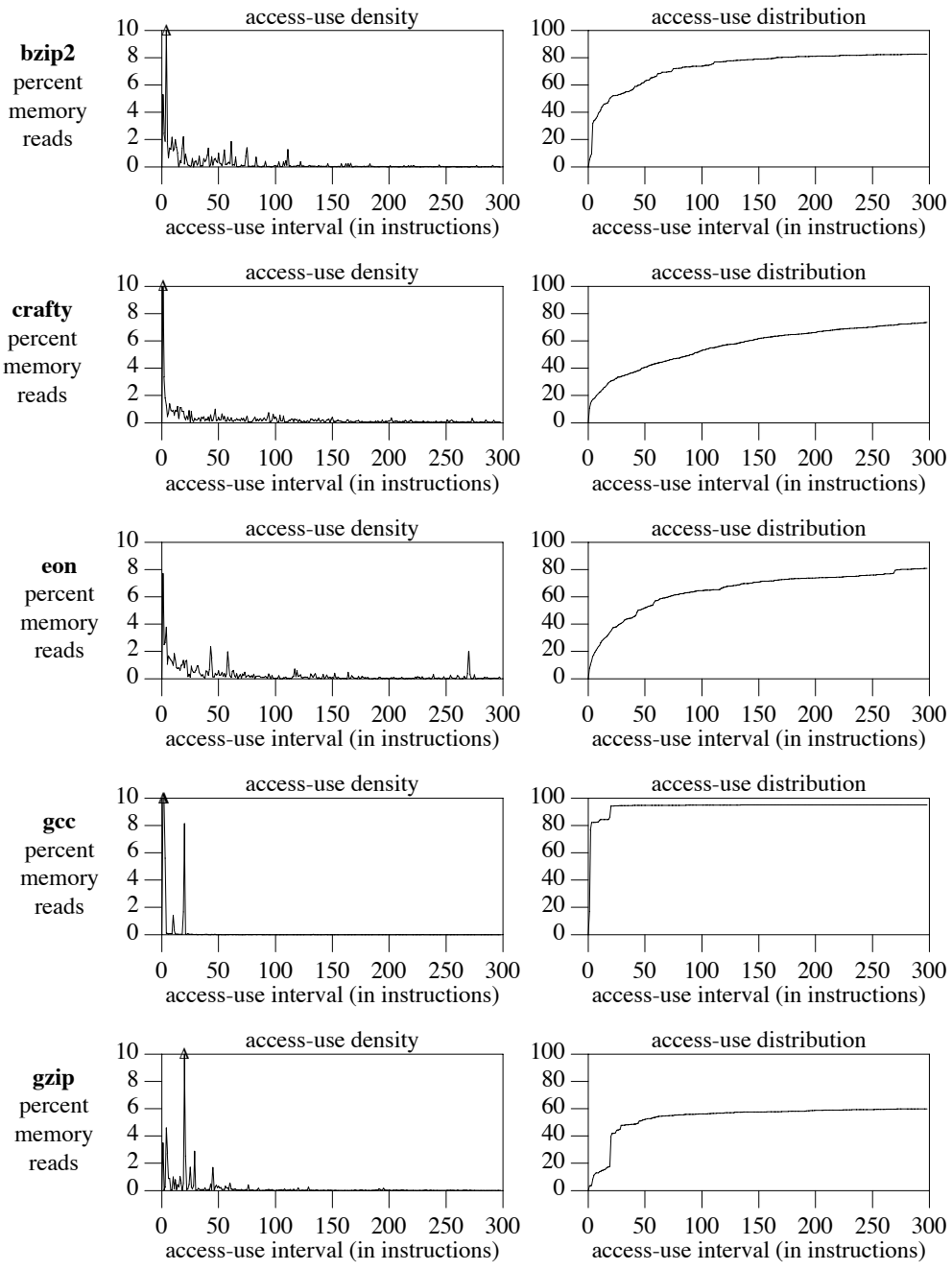

Figure 9.8: *Memory Access-Use Intervals.* Data results for the BZIP2, CRAFTY, EON, GCC, and GZIP programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

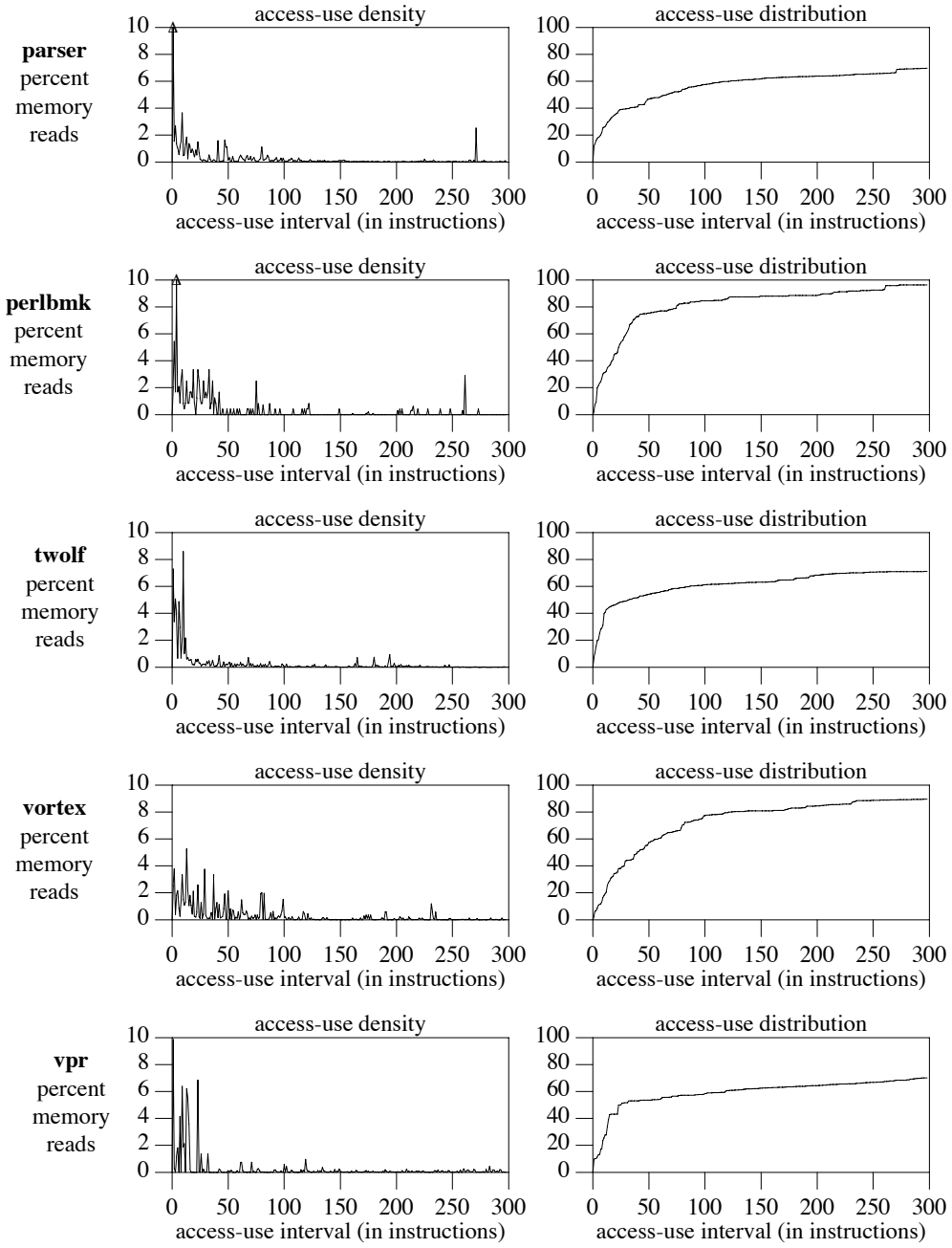

Figure 9.9: *Memory Access-Use Intervals.* Data results for the PARSER, PERLBMK, TWOLF, VORTEX, and VPR programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

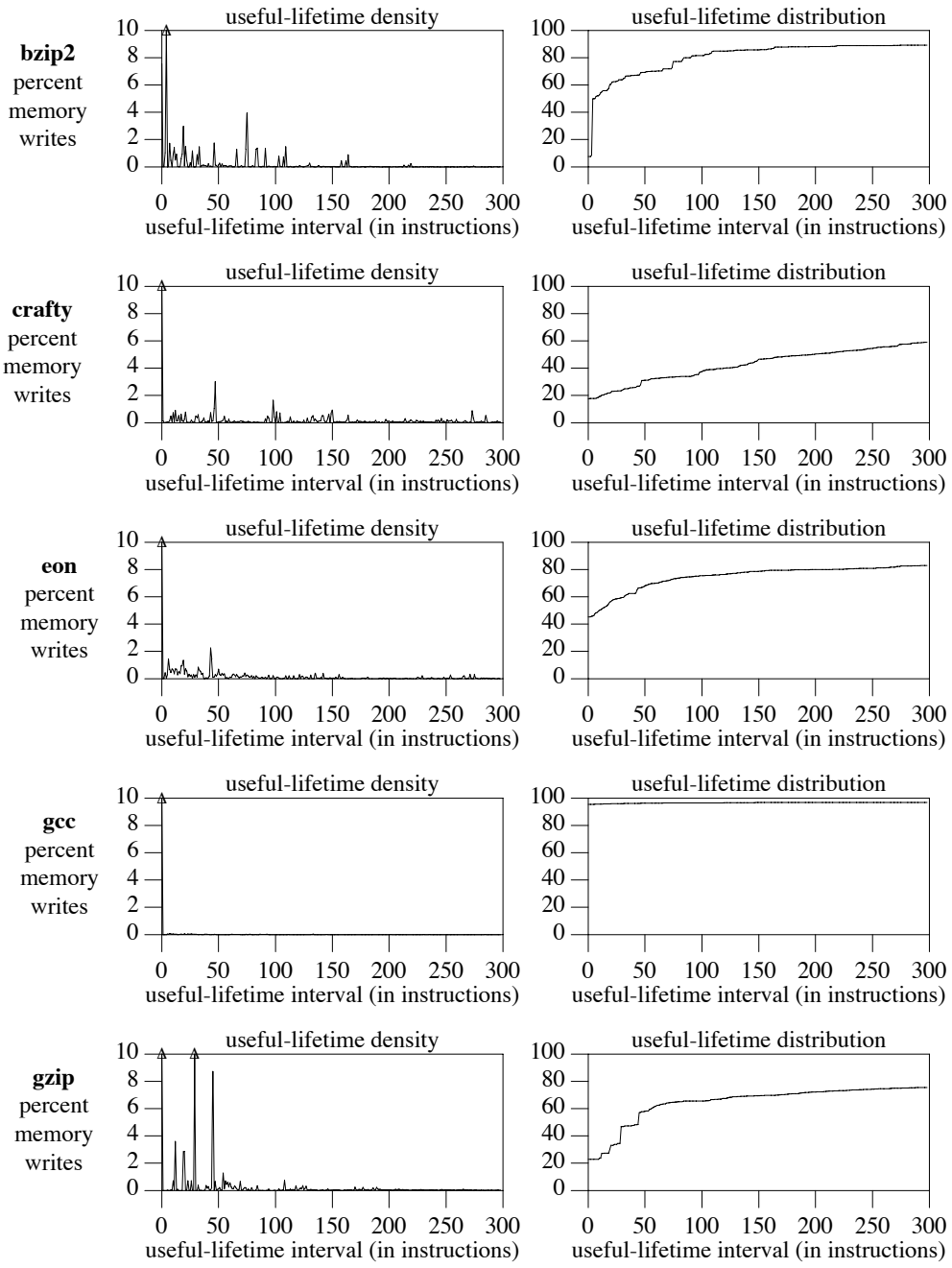

Figure 9.10: *Memory Def-Last-Use Intervals.* Data results for the BZIP2, CRAFTY, EON, GCC, and GZIP programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

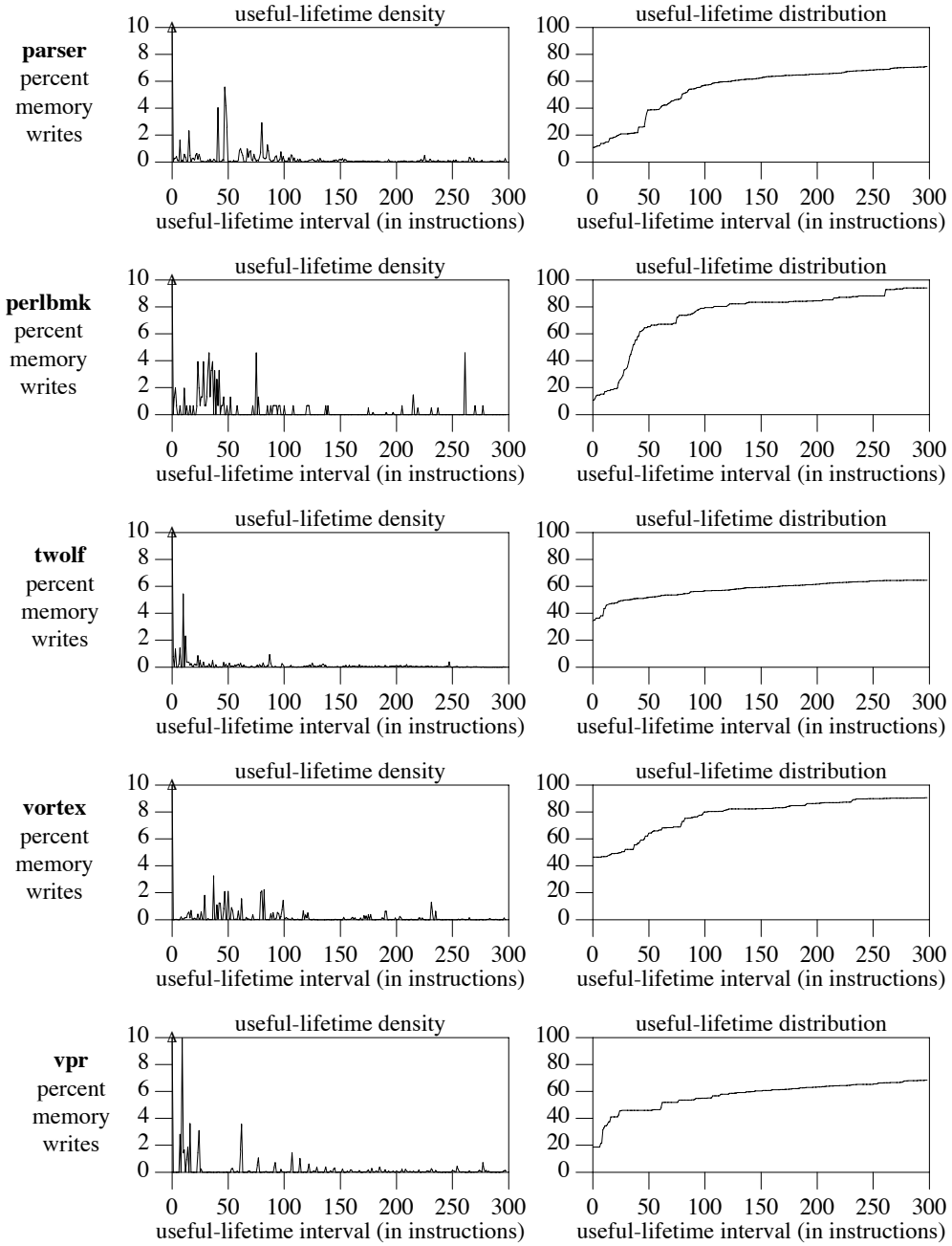

Figure 9.11: *Memory Def-Last-Use Intervals.* Data results for the PARSER, PERLBMK, TWOLF, VORTEX, and VPR programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

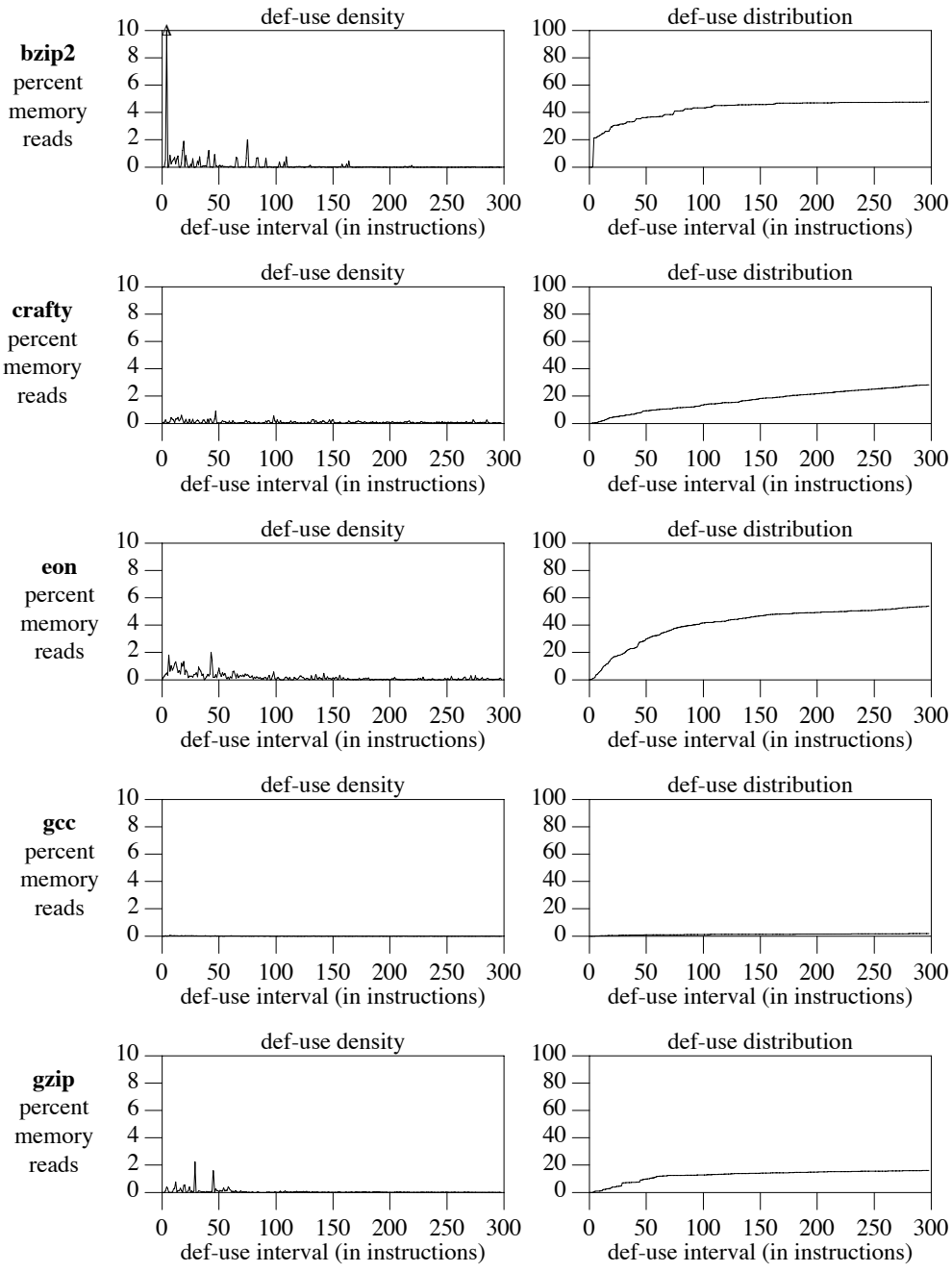

Figure 9.12: *Memory Def-Use Intervals.* Data results for the BZIP2, CRAFTY, EON, GCC, and GZIP programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

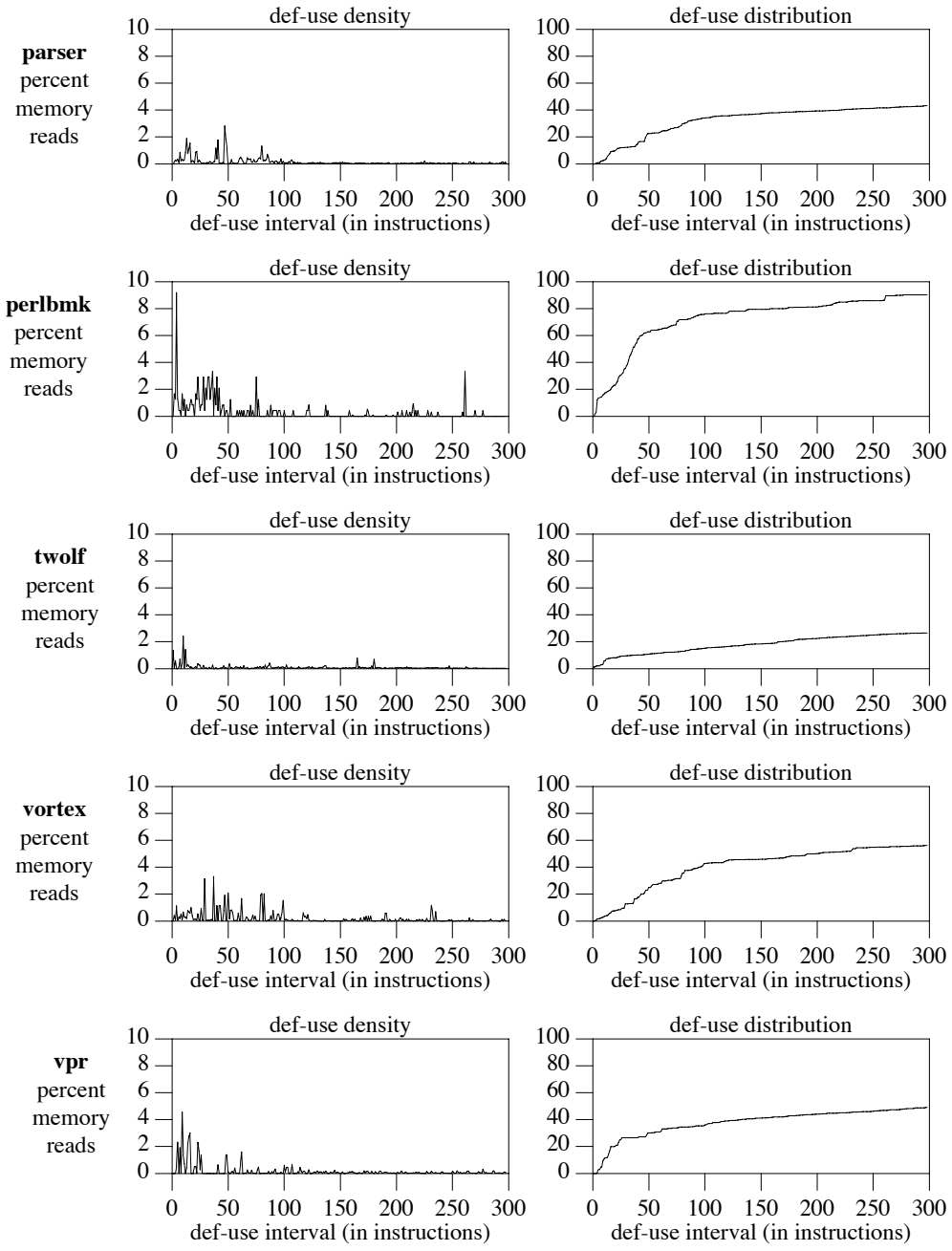

Figure 9.13: *Memory Def-Use Intervals.* Data results for the PARSER, PERLBMK, TWOLF, VORTEX, and VPR programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

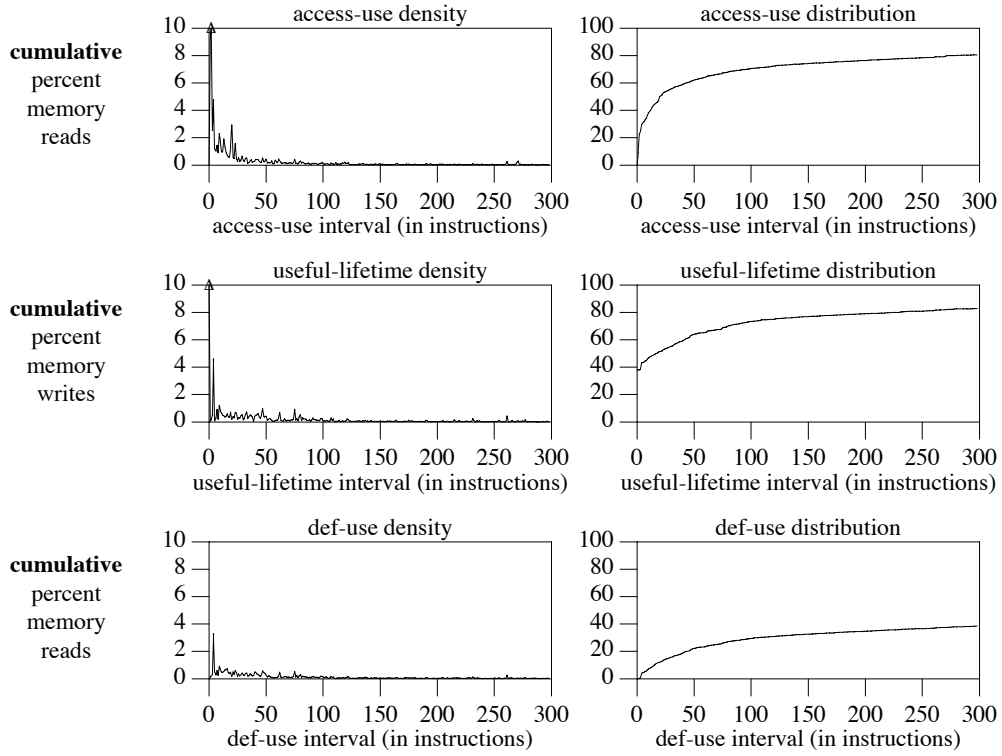

Figure 9.14: *Cumulative Memory Intervals over all benchmarks.* The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

operands in a scaled Resource Flow microarchitecture using memory filter units (of Chapter 8). They can acquire their operands in any of the following ways (and due to dynamic re-executions, possibly more than a single way within the same instruction execution residency of an IS):

- from the L1 cache (or higher in the memory hierarchy)
- directly from the originating write operation within the execution window
- from the originating write operation but by traversing one or more memory filter units
- from a preceding memory filter unit that has cached the operand

We should remember that only a read operation its desired operand from the L1 data cache (or beyond) is truly bad (relatively speaking) as compared with the other alternatives. The the case of acquiring an operand directly from the defining instructions, the def-use data is most appropriate.

But when filtering (and associated caching of operands) is available, as we expect it to be in our scaled Resource Flow microarchitecture), then the access-use interval data is more appropriate. Only upon failing to acquire an operand directly from the defining instruction or from a cached copy, does the need for going out to the convention memory hierarchy (L1 and beyond) come into play.

Returning to the data, it still shows that our plan for scaling the execution window through the segmentation of the operand transfer buses is still a valuable microarchitectural feature for performance reasons. Focusing on the consequences of memory writes (and referencing Figures 9.10, 9.11, 9.14), we see generally that about 60/useful-lifetimes of write operations only need to be transported 50 instructions into the future in order to encounter a waiting read operation of the same variable. This shows us that for execution windows of approximately double this number (or about 100 instructions) in size, these subsequent memory reads do not have to access the L1 cache in order to be satisfies. Rather they will be satisfied from the previous write directly using the operand transfer fabric. If an operand transfer bus span of 50 to 100 instructions in length is chosen, then this read operation is satisfied without the memory operand even having to traverse a memory filter unit (all the better). For those memory write operations that do not have a corresponding read within the size of the execution window as a whole, remember that the operand still has the chance of being cached within at least one memory filter unit due to their L0 caches (reference Chapter 8). Looking now at the situation from the point of view of a memory read operation, its prospects of acquiring its operand are given by the access-use data. This data actually forms something of a conservative estimate of the expectation for acquiring the desired operand within the confines of the execution (avoiding a costly trip to the L1 cache or beyond). For reads (and referencing Figures 9.8, 9.9, 9.14), the access-use data alone tells us that we can expect to receive our desired operand in about 60% of the time from within the preceding 50 instructions, and for about 55% of the time from within the preceding 25 instructions. This is still a pretty good deal considering that trips to the higher memory hierarchy (from L1 and beyond) are often quite expensive.

9.3 Summary and conclusions

We have presented data for various access intervals associated with both register and memory variable instances. The data for the register intervals is consistent with and confirms the prior work by Franklin et al. That data shows that for most general purpose sequential program codes (for example SpecINT) as many as approximately 70% of the register reads (a use of the variable instance) will have had their variable instances defined within the preceding 25 dynamic instructions (this is drawn from the def-use results). This indicates that for large instruction window sizes (where 25 is a small fraction of the size) that there is a good chance that the associated register operand

can come directly from the defining instruction without having to be first stored in either the architected register file or some other centralized resource (such as a future file). When some form of register buffering or caching is employed (as is the case with register filtering of Chapter 8), it can be expected that the percent of register reads satisfied within the execution window will be even higher, and could average about 95% of all reads having been either defined or present in an operand cache located within the last 25 dynamic instructions (from the access-use interval data).

For memory variable instances, there is not as much likelihood of any given memory read (*load instruction*) having its variable instance already within the execution window as compared with the register case. For memory reads, the cost of having to go outside the execution window of the processor is higher than as for registers and any possible operand bypass of either the load-store queue or the L1 data cache is still welcomed. Our data shows that about 30% of memory read operations can expect to have their preceding instance definition within the last 100 dynamic instructions (def-use characterization results). Although this is not as useful for operand bypass, as in the case of registers, any bypass that can occur in these cases reduces the access burden on both the load-store queue as well as the L1 data cache. Similarly to the situation for registers, greater percentages of reads (memory load requests) can be expected to be satisfied within the execution window alone (bypassing the LSQ and L1 cache) when distributed caches (distributed L0 caches) are employed. From the access-use interval characterization data, about 55% of all memory loads could benefit by operand bypass of centralized resources for machines that can have the same previous 100 dynamic instructions within the window.

This data (and especially the register access interval data) supports and validates our strategy for scaling our Resource Flow machine of Chapter 4 (OpTiFlow) by segmenting the operand transfer buses that interconnect the issue stations into lengths of between something and something. The very task of implementing one such scaled microarchitecture is presented in the next chapter.

Chapter 10

The Levo microarchitecture

In previous chapters we presented the basic mechanisms for managing a much larger amount of instruction level parallelism (ILP) than is structurally possible in most existing microarchitectures. Chapter 3 presented the basic concepts embodied in Resource Flow computing and Chapter 4 introduced a representative microarchitecture implementing those concepts. But having the mechanisms available to manage large numbers of machine resources with regard to ILP does not in itself address how a machine might be physically scaled in order to overcome hardware implementation restraints such as those associated with wire length, signal integrity decay, and signal delay.

In this chapter we combine the ideas of Resource Flow computing (many of them already presented in the OpTiFlow microarchitecture) with the idea of operand forwarding units (and more specifically operand filtering units, presented in the previous chapter) to come up with a possible microarchitecture that embodies most of the advantages of the OpTiFlow microarchitecture along with the ability to be physically scalable. This new microarchitecture will allow for the implementation of much larger physical sizes and numbers of machine resources. The possible machine size, in terms of resource components, with this present example are currently substantially beyond those that can be achieved using any conventional microarchitectural models. This microarchitecture will also include the idea of dynamic instruction predication (presented in Chapter 7) as well as a form of multipath execution (the Disjoint Eager Execution type) that was briefly described in Chapter 2. The basic framework of this new proposed microarchitecture is termed *Levo*.

The remainder of this chapter is organized as follows. We first provide an overview of the microarchitecture to show both its similarity to conventional machines at a high level, but also to orient the reader to where the major innovations in this microarchitecture occurs. This is followed by a more detailed discussion of the most innovative part of the microarchitecture; namely, where instructions are staged for execution and how execution and re-execution is managed. The components and mechanisms used for entering instructions into execution forms the heart of this

microarchitecture and constitutes its major departure from both conventional and other proposed research machines. We then discuss how the various components of the microarchitecture related to instruction execution are interconnected so as to provide physical scalability. In the next section we discuss how persistent machine state is handled within the microarchitecture. The persistent state in view consists of architected machine registers, temporarily pending memory writes as well as other cached memory values, and microarchitectural specific state that is related to the microarchitectural instruction predication scheme that is employed. As will be shown, due to the distributed nature of the microarchitecture, persistent machine state is not stored in a single machine component but is rather rotated among various components in a rotating fashion. We then discuss in more detail some additional operational characteristics of the machine. These include how precise interrupts and exceptions may be handled, the fetch heuristics of the machine, and how multipath execution is performed. Next, we present our simulation methodology and some results from simulation. The goal is to provide some preliminary expectations of the performance of the microarchitecture in various configurations. Our intention with simulation is to both compare our new microarchitecture with a conventional one as well as to explore some design-space variations of the new microarchitecture. And finally we summarize in the last section.

10.1 Overview

This microarchitecture is very aggressive in terms of the amount of speculative execution it performs. This is realized through a large amount of scalable execution resources. Resource scalability of the microarchitecture is achieved through its distributed nature along with repeater-like components that limit the maximum bus spans. Contention for major centralized structures is avoided. Conventional centralized resources like a register file, reorder buffer, and centralized execution units, are eliminated.

As with the OpTiFlow microarchitecture (described in Chapter 4), this microarchitecture also shares some similarity with conventional microarchitectures in many respects. As with OpTiFlow, this microarchitecture can also implement any instruction set architecture (ISA), either a legacy or new one. This is an important characteristic keeping this microarchitecture relevant since certain legacy ISAs are not going away (x86) and also because new ISAs are not very common. The common aspects of this microarchitecture are discussed here and then the distinctives of it are presented in the subsequent section. The L2 cache (unified in the present case), and the L1 instruction cache are both rather similar to those in common use. Except for the fact that the main memory, L2 cache, and L1 data cache are generally all address-interleaved, there is nothing further unique about these components. The interleaving is simply used as a bandwidth enhancing technique and is not functionally necessary for the design to work.

The microarchitecture also addresses several issues associated with conditional branches. Alternative speculative paths are spawned when encountering conditional branches to avoid branch misprediction penalties. Exploitation of control and data independent instructions beyond the join of a hammock branch [31, 150] is also capitalized upon where possible. Choosing which paths in multipath execution should be given priority for machine resources is also addressed by the machine. The predicted program path is referred to as the *mainline* (ML) path. Execution resource priority is given to the mainline path over other possible alternative paths that are also being executed simultaneously. Since alternative paths have lower priority for resource allocation than the mainline path, they are referred to as *disjoint* paths. This sort of strategy for the spawning of disjoint paths results in what is termed *disjoint eager execution* (DEE). Disjoint alternative execution paths are often simply termed *DEE paths*. These terms are taken from Uht. [148] Early forms of this microarchitecture were first presented by Uht [154] and then Morano. [96] Additional detail on this microarchitecture can also be found in work by Uht et al. with the same name (Levo). [153]

The I-fetch unit first fetches instructions from i-cache along one or more predicted program paths. Due to the relatively large instruction fetch bandwidth requirement, the fetching of several i-cache lines in a single clock may be generally required. Instructions are immediately decoded after being fetched and any further handling of the instructions is done in their decoded form. Decoded instructions are then staged into the *instruction dispatch buffer* so that they are available to be dispatched into the *execution window* when needed. The execution window is where these microarchitectures differ substantially from existing machines. This term describes that part of the microarchitecture where the issue stations and processing units are grouped. The instruction dispatch buffer is organized so that a large number of instructions can be broadside loaded (dispatched) into the issue stations within the execution window in a single clock.

Figure 10.1 provides a high-level view of this microarchitecture. At this level of detail, this is essentially identical to the OpTiFlow microarchitecture presented in Chapter 4.

Multiple branch predictors have been used in the I-fetch stage so that several conditional branches can be predicted in parallel in order to retain a high fetch and dispatch rate. Research on the multiple simultaneous branch prediction and dispatch is ongoing. As we will see shortly, the machine is organized into *columns* of issue stations. The goal of the machine is to dispatch instructions from the dispatch buffer to all of the issue stations within a single column of issue stations within a single clock period. Although it is possible to dispatch less instructions than the number of issue stations in a column, this results in lower issue station utilization and lower execution performance. Instructions can be dispatched to issue stations with or without initial input source operands. Instructions can also be dispatched with predicted input operands using value prediction. Instructions are dispatched in-order.

When a new instruction is dispatched to an issue station and once it has acquired its input

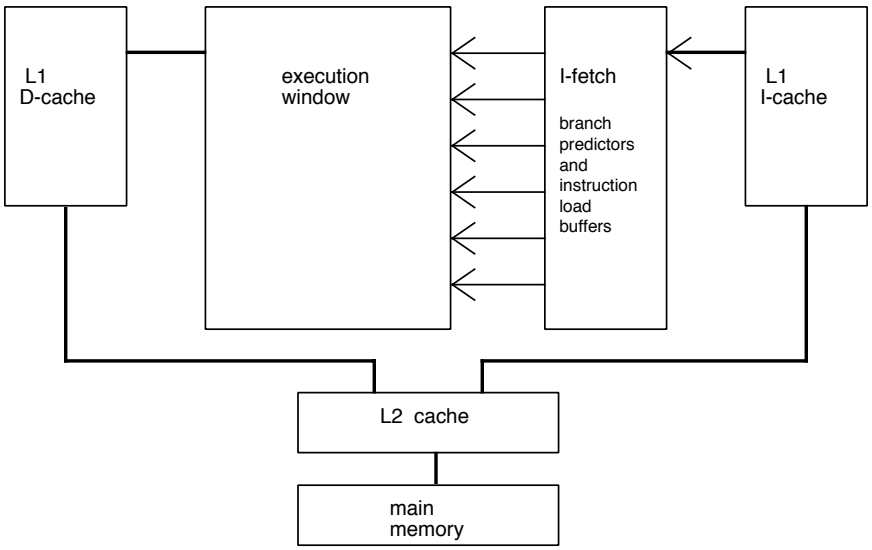

Figure 10.1: *High-level view of a Resource Flow microarchitecture.* Shown are the major hardware components of the microarchitecture. With the exception of the execution window block, this is similar to most conventional microarchitectures.

operands (if it was not dispatched with predicted ones) it begins the process of contending for execution resources (depending on microarchitectural implementation design point). Operand dependency determination is not done before either instruction dispatch (to an issue station) or before instruction operation issue to an execution unit. All operand dependencies are determined dynamically through snooping after instruction dispatch. As stated previously, each instruction remains in the issue station (IS) to which it was dispatched, executing and possibly re-executing until it is ready to be retired (either committed or abandoned). All retirement occurs in-order.

Before more operational details about of the microarchitecture can be discussed, we need to look into the real heart of the machine – its execution window. This is taken up in the next section.

10.2 The execution window

Figure 10.2 shows a more detailed, but still very much simplified, view of the execution window of this microarchitecture and its subcomponents.

Several ISes may share the use of one or more execution units. The execution units that are dispersed among the ISes are termed *processing elements* (PEs). Each PE may consist of an unified all-purpose execution unit capable of executing any of the possible machine instructions or, more likely, consist of several functionally clustered units for specific classes of instructions (integer ALU, FP, or other). As part of the strategy to allow for a scalable microarchitecture (more on

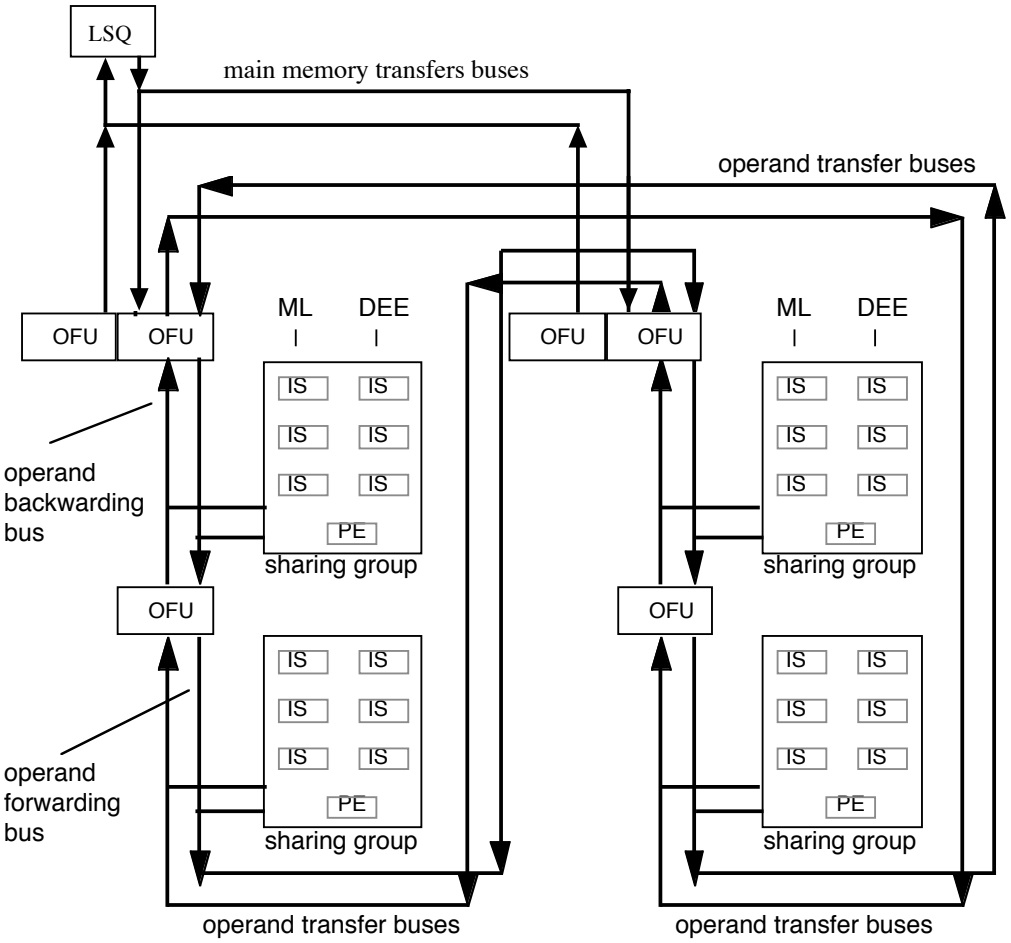

Figure 10.2: *The Execution Window of a distributed microarchitecture.* Shown is a layout of the Issue Stations (IS) and Processing Elements (PE) along with some bus interconnections to implement a large, distributed microarchitecture. Groups of ISes share a PE; each group is termed a *sharing group*.

scalability is presented below), the ISes are laid out in silicon (assuming a single substrate for our present purposes) on a two-dimensional grid whereby sequentially dispatched instructions will go to sequential ISes down a column of the two-dimensional grid of ISes. The use of a two-dimensional grid allows for a design implementation in either a single silicon integrated circuit (IC) or through several suitable ICs on a multi-chip module. The number of ISes in the height dimension of the grid is termed the *column height* of the machine and also is the upper limit of the number of instructions that can be dispatched to ISes in a single clock.

A group of issue stations along with their associated PE is called a *sharing group* (SG). As the name suggests, all of the ISes within a sharing group share the same execution resource within that group. Although we have only examined using a single processing element within each sharing group, arrangements with more than a single processing element per sharing group are also possible. Sharing groups have a relatively high degree of bus interconnectivity within them, much as conventional microarchitectures have between their similar components. The transfer of a decoded instruction, along with its associated operands, from an IS to its PE is isolated to within the SG they belong to. The use of this execution resource-sharing arrangement also allows for reduced interconnections between adjacent SGs. As will also be discussed later, this feature of SGs is a critical element of the ability of the microarchitecture to be physically scalable. Basically, only operand results need to flow from one SG to subsequent ones. in particular, instruction operation issue from an IS to a PE is entirely within the SG, as is the transfer of the PE result operand back to the IS. Sharing groups can be thought of having high internal cohesion (especially within the ISes) while having low coupling with other sharing groups. As part of the ability to support multipath execution, there are two columns of ISes within each SG. The first IS-column (the one on the left side of the SGs in the figure) is reserved for the mainline path of the program and is labeled *ML* at the top of the appropriate IS columns in the figure. The second column of ISes (on the right side of each SG) is reserved for the possible execution of a DEE path and is labeled *DEE* at the top of the appropriate IS columns in the figure. Other arrangements for handling multipath execution are possible but have not been explored.

The example machine shown in Figure 10.2 has a column height of six, and is shown with six instruction load buses feeding up to six ISes in the height dimension. Each SG contains three rows of ISes (for a total of six) and a single PE. Overall this example machine consists of two columns of SGs, each with two SG rows. A particular machine is generally characterized using the 3-tuple:

- sharing group rows
- issue station rows per sharing group
- sharing group columns

These three characteristic parameters of a given machine are greatly influential to its performance, as expected, and the 3-tuple is termed the *geometry* of the machine. These three numbers are usually concatenated so that the geometry of the machine in Figure 10.2 would be abbreviated *2-3-2*. The set of mainline IS columns and DEE IS columns should be viewed as two sets of columns (one overlaid onto the other) that are sharing common execution resources and a common interface to the operand transfer interconnection fabric. Each set may be operated on rather independently based on the strategy (or strategies) employed for the management of alternative DEE paths. Also shown in the figure are two sets of buses to transfer operands and requests for operands among the issue stations. Each set of buses is represented by a bold line in the figure (representing a set of individual buses). One set of buses allows for operands to be forwarded (travel forward in program-ordered time), while the other set allows for requests for operands to be backwarded (travel backwards in program-ordered time). Both the forwarding buses and the backwarding buses are adjacent (in the figure) and both are periodically segmented through the use of the operand filtering units (labeled OFU in the figure). Arrows on the buses show the direction of transfers. Each OFU is actually a combination of a memory filtering unit (MFU), a register filtering unit (RFU), and a predicate filtering unit (PFU). These filter units were discussed in more detail in Chapter 8. We take a special note here that at the top of each column of ISes there are two OFUs separating the forwarding and backwarding buses. The need for two sets of OFUs is discussed in a subsequent section. More information on these operand transfer buses and the OFUs is also given in the next section.

When an entire column of mainline ISes are free to accept new instructions, up to an entire column worth of instructions are dispatched in a single clock to the free column of ISes within that SG column. Newly dispatched instructions always to the column of ISes reserved for mainline execution. The same column of instructions may also get dispatched to the DEE column if it is determined advantageous to spawn an alternative execution path at dispatch time. Different methods for dispatching alternative path instructions to DEE columns of ISes are possible (including dynamically after initial dispatch) but these are beyond the scope of the present discussion. An attempt is made by the I-fetch unit to always have an entire column's worth of decoded instructions ready for dispatch, but this is not always possible. If less than a whole column of instructions are dispatched, those ISes not receiving an instruction are essentially unused and represent a wasted machine resource. Conditional branches are predicted just before they are entered into the instruction dispatch buffer and the prediction accompanies the decoded instruction when it is dispatched to an IS. Again, instructions remain with their IS during execution and possible re-execution until they are ready to be retired. Unlike the proposed OpTiFlow microarchitecture, here an entire column of ISes gets retired at once (not at the granularity of a single IS). As a column of ISes gets retired, that column becomes available for newly decoded instructions to be dispatched to it. It is

also possible for more than a single column to retire within a single clock and therefore for more than a single column to be available for dispatch within a single clock. However this is not likely and research efforts so far have not yielded a machine capable of such high execution rates.

All of the ISes within an IS column (either a mainline column or a DEE column) are assigned a contiguous set of time-tag values. The IS at the top of the column has the lowest valued time-tag within it, and the value is increment by one for successive ISes down the column. The topmost IS of an adjacent IS mainline column will contain a time-tag value one higher than the previous column's IS having the highest valued time-tag (which will be located at the bottom of the previous column). The time-tag value assigned to ISes within a row and jumping from one SG column to an adjacent one (one mainline column of ISes to another in an adjacent SG column) have time-tag values differing by the height of a column as counted in ISes. At any one point in the machine operation, one column serves as the *leading* column with the IS having the lowest valued time-tag (generally zero) within it, while another column serves as the *trailing* column with the IS having the highest valued time-tag. The lead IS column contains those instructions that were dispatched earliest while the trailing IS column contains those instructions most recently dispatched.

It will be the leading mainline IS column that is always the one to be committed next. Commitment of a column occurs when all of the individual issue stations within the column are ready for commitment. Each individual issue station is ready for commitment when it has executed its instruction at least once, has forwarded all of its output operands, and has not received any new operands for which it has not already executed or re-executed with, and whose previous issue station (if not at the top of the column) is also ready to commit. These conditions will eventually converge allowing for commitment of each SG column in turn. When a column of SG

Upon a commitment of the leading column, the time-tags within ISes and operands within the machine are effectively decremented by an amount equal to the height of the machine in ISes (or column height). All time-tags can be decomposed into row and column parts. The column part of the time-tag serves as a name for the logical column itself. When the leading column is retired, all other columns effectively decrement their column name by one. This effectively decrements all time-tag within each column by the proper amount and serves to rename all of the operands within that column. With this column-row time-tag decomposition, the row part of the tag is actually fixed to be the same as the physical row number in the physical component layout in silicon. This means that upon column retirement, the column part of any time-tag is decremented by the number of columns being retired (generally just one). The next IS column in the machine (with the next higher time-tag name) becomes the new leading column and the next to get retired. The operation of decrementing column time-tags in the execution window is termed a *column shift*. The term was derived from the fact that when the whole of the execution window (composed of ISes and PEs) is viewed as being logically made up of columns with the lowest time-tag value one always at the far

left, the columns appear to shift horizontally (leftward) on column retirement as if the components made up a large shift register, with each column of machine components being a single stage, and shifting right to left. In reality, columns are simply renamed when the column part of the time-tags are decremented, and the logical columns can be thought of as being rotated left.

10.3 Bus interconnect and machine scalability

As with the OpTiFlow microarchitecture of Chapter 4, some sort of interconnection fabric is used to allow for the backward requesting of operands among ISes as well as for the forwarding of result operands. Generally, several operand transfer buses are used in parallel to handle the operand forwarding bandwidth requirement. These parallel bundled buses are shown in Figure 10.2 as bold lines (consistent with industry usage for the same). In the present microarchitecture, all operand transfer buses are multiplexed for use by all operand types, but other arrangements are certainly possible. Multiplexing of all operands onto a single set of buses provides the best bandwidth efficiency but represents just one possible arrangement. However, separate buses are still maintained for transfers in two directions as related to program-ordered time: forward and backward.

The set of operand transfer buses, partitioned into segments, and separated by operand forwarding units (OFUs in the figure). Two types of operand transfer buses are distinguished in the figure. These are termed *operand forwarding buses* and *operand backwarding buses* respectively. For each adjacent pair of operand transfer buses, when the buses are shown vertically in Figure 10.2 on the immediate left side of the columns of SGs (only two SG columns in the figure) the backwarding bus is on the left of the pair and the forwarding bus is on the right. The arrows on the backwarding buses are pointing upwards and visa versa for the forwarding buses.

In order to make the machine scalable to large sizes, all operand transfer buses are of a fixed (constant) length, independent of the size of the machine (in numbers of components, ISes and PEs). This is accomplished through the use of OFUs already discussed. The connectivity represented by the combination of the operand transfer buses and the OFUs loops around the whole of the execution window of machine forming a closed path. This closed path somewhat resembles the register operand transfer bus of more conventional microarchitectures, like the Common Data Bus in the original Tomasulo design using reservation stations [141] but also found in other proposed machines. [104] The set of SGs, operand transfer buses, and OFUs in the figure represent a logical arrangement only and does not necessarily represent how all components might be laid out in silicon. Since each SG has only a single point of connection with the operand transfer buses, these buses are only segmented by OFUs at the granularity of an SG. Although there need not be a one-to-one relationship between the number of SGs and OFUs in any given machine (as happens to be shown in Figure 10.2), there always needs to be OFUs present at the top of each column. This is

due to the fact that this microarchitecture performs instruction commitment at the granularity of whole columns. With regard to time-tags, all OFUs logically have the same time-tag values as that of the oldest IS within the SG it is adjacent to. As the machine commits a whole column of SGs, all of the time-tags in the machine (inside ISes, OFUs, et cetera) get effectively decremented by the number of ISes in an SG column (historically called a shift operation). This shift operation will leave one IS within an SG at the top of a column with the time-tag value zero. The current IS with the time-tag value of zero represents the oldest instruction remaining within the execution window of the machine. The OFU that is associated with the SG containing the IS with the time-tag value of zero will correspondingly contain the architected register and predicate state that is not in the process of being changed through subsequent speculative instructions in the execution window. As with ISes, OFUs (its component filter parts) have time-tags associated with them. All OFUs in a machine will always have the same time-tag value as the top-most (oldest) IS contained within the SG that the OFU is adjacent to.

As was discussed in the previous chapter, the OFUs each contain am MFU for memory, a RFU for registers, and a PFU for predicates. These filtering units (one for each type of operand) act similarly to caches, each serving for the type of operand it is designed for. As ISes make requests for operands (be it of whatever type) the request eventually gets placed onto an operand backwarding bus. This request then travels backwards (as compared with program-ordered time) among the various components within the execution window. This request may be serviced by either ISes or by the filtering unit for the type of operand being requested. the first opportunity for requests to be satisfied always comes from those ISes that are located within the same SG as the requesting IS. However, requests are always (as least presently) still transferred backwards even if the request may be satisfied by an IS in the local SG. The reason is due to the fact that no knowledge of operand dependencies among instructions is statically maintained so it is not known by the requesting IS where that request will be satisfied from. As the operand request travels backward to previous (program-ordered) ISes, it may be satisfied by one of those ISes. However the request will also eventually hit an OFU. At the very least, an OFU is always located at the top of each column and a request will always be intercepted by that OFU at a minimum. In any case, when an OFU receives an operand request, it may satisfy the request itself. In the case of a register request, it will often be satisfied by the first RFU encountered. This is due to the fact that all RFUs can contain a possible value for each architected register. The only reason that an RFU might not contain a register value is if that register was nullified (as discussed in Chapter 3). In most cases, an RFU will have a value (albeit a speculative value) for a register operand and the resulting operand will be forwarded so that it will eventually read the originating IS to satisfy it. In the case of memory operands, there is less likelihood of a request being satisfied by any single MFU. This is because the address space of memory and the set of active memory locations is much larger than

that of the architected register address space. The situation for predicates is somewhat similar to that for registers since the address space for predicates is only equal to the number of ISes within the execution window. When an operand request is satisfied by an OFU, it is termed a *hit* (to borrow from the cache metaphor). If an OFU cannot satisfy a request, it is termed a *miss*. On hits, requests for operands are not further backwarded (there is no need). However on misses, the request is further backwarded so that ISes and OFUs located further back in program-ordered time also get a chance to satisfy the request. For register and predicate operands, at a maximum requests only need to be backwarded to that OFU that currently has the time-tag value of zero. This OFU corresponding to the one that presently holds the current architected machine state and so there is no further resort for finding possible values. However for memory operands, the architected state is stored in the remainder of the memory hierarchy so the request needs to be somehow further backwarded.

For register and predicate operand requests, once they are satisfied (always within the execution window) they are then forwarded using the operand forwarding bus fabric. Forwarded operands also travel through the OFUs as the OFUs segment the entire forwarding fabric. These forwarded operands can cause updates to the storage within either RFUs or PFUs (according to operand type). This roughly corresponds to a cache update operation in the case of memory operands. Eventually, a forwarded operand will reach the IS that originally made the request and it will be snarfed by that IS.

A memory operand request that travels back through that MFU that currently has the time-tag value of zero continues backward to the L1 data cache. This is accomplished by the operand being switched onto another set of buses (those buses connecting to the L1 data cache). Since all of the MFUs (within OFUs) that are physically located at the top of each column of SGs may, in turn, have the time-tag value of each. All MFUs at the top of the SG columns have connectivity to the buses connecting to the L1 data cache. The set of buses connecting the MFUs at the top of the SG columns to the L1 data cache is called the *memory operand transfer buses*. These are multimaster buses and are all bidirectional in the present microarchitecture. Optionally separate buses can be provided for transfers in each direction. Once a memory request reaches back to the L1 data cache, it is handled as it would be in a convention machine with a convention memory hierarchy. Specifically, the request may be serviced by the L1 data cache or may miss and need further processing at higher levels in the hierarchy. Memory requests returning from the L1 data cache enter the MFUs at the top of the SG columns and are forwarded from there. Eventually, the IS that made the original request will snarf the operand finally satisfying its request.

10.4 Rotating and persistent machine state

In this section we take a little bit more of a detailed look at how machine state is maintained during its operation, and especially in light of when column shifts occur. Some of this state is persistent architected machine state (this would consist of the machine architected registers) but the remainder of this state is speculative state that needs to be maintained (tracked) through column shifts also. There is no centralized storage for any of this state. Rather, it is stored alternatively in various hardware structures that rotate as the machine shifts columns. Persistent register, predicate, and some persistent memory state is stored in the filter units. Persistent state is not stored indefinitely in any single filter unit but is rather stored in different units as the machine executes column shift operations (columns of ISes get retired). However, this is all quite invisible to the ISA. Although these units are fixed in silicon, they get renamed logically in order to effect the logical shifting (or rotating) of the machine columns.

A very simple machine example can illustrate how the persistent state rotates (logically) within the machine as column shifts (retirements) occur. Figure 10.3 shows a simple Levo-type machine microarchitecture with two columns and four ISes per column (sharing groups are not shown for clarity). The operand filter units (labeled OFU) are shown at the top of each column. Each operand filter unit actually consists of individual subunits for register, memory, and predicate filters. Two operand filter units are shown at the head of each column. In any machine implementation, operand filter units are minimally required at the top of each column in order to provide for persistent state throughout machine operation. Operand filter units elsewhere within machine columns may be needed to maintain constant operand bus spans (length of the buses) but they are not required for the management of the persistent machine state. The operation of the operand filter units that head each machine column will become clearer as we examine the machine operation through column retirement (consisting of instruction commitments or abandonments) and associated machine shifts. Any operand filter units besides the ones at the top of the columns are not shown in both this figure and our subsequent example figures since they are not involved in the retention of persistent machine state through column shift operations. Also shown is the load-store-queue (LSQ) that is situated between the memory filter units (subsumed within the operand filter units in our current example) and the rest of the memory hierarchy starting with the L1 data cache (also not show in present example). Speculative memory write operands are stored in the memory filter units (MFU) in a rotating manner, but only committed memory write operands are transferred to the load-store-queue. The existence of the LSQ allows for buffering between instruction commitment (and particular memory write instructions) and the rest of the memory hierarchy. Without this buffering of memory write operands, the commitment of memory write instructions would have to be stalled until a clear path was available to write the associated value to the memory system. There the LSQ

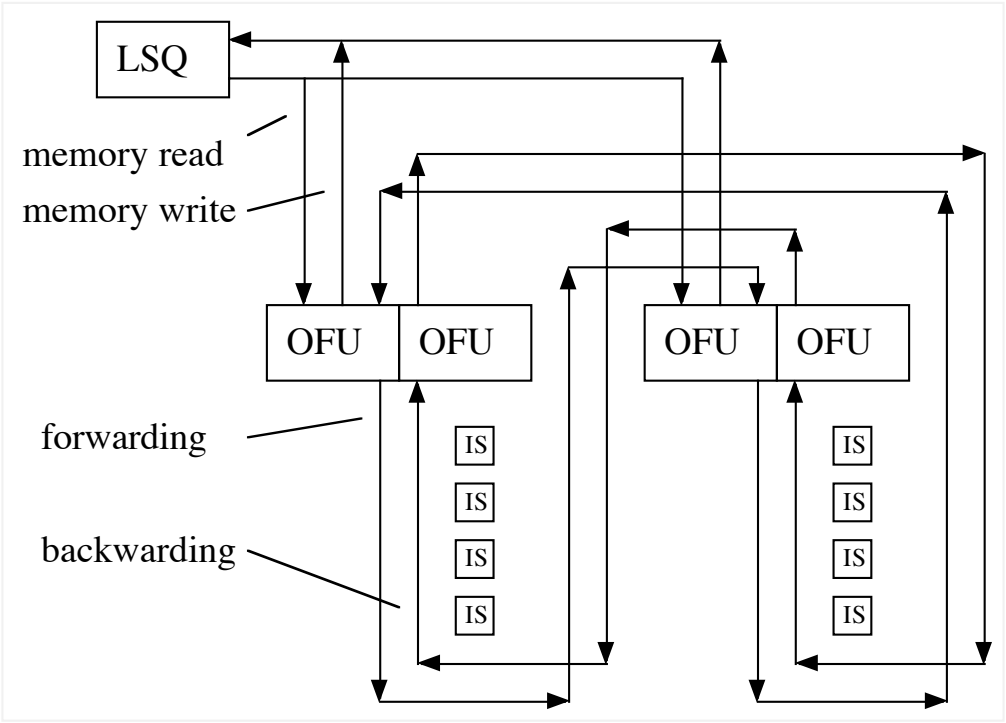

Figure 10.3: *An example Levo microarchitecture to illustrate rotating machine state.* Two columns of issue stations are shown along with their operand filter units at the head of each. Both forwarding and backwarding operand transfer buses are shown, along with memory read and write buses to the load-store-queue (LSQ).

in this microarchitecture serves the same purpose as it does in conventional microarchitectures. The LSQ needs to be connected to all of the MFUs within the microarchitecture since any one of them (in a rotating fashion) will be the one with the committed memory operands that need to be stored to the memory hierarchy through the LSQ. This will become clearer as we unfold this example. The machine shown in Figure 10.3 is similar to that of Figure 10.2) but will serve as our starting point for giving some additional detail on how the persistent machine state is handled during its operation.

For our present purposes, we will now further simplify the microarchitecture of Figure 10.3 while adding some labeling to help us. The simplification is shown in Figure 10.4. In this figure (Figure 10.4), the operand backwarding buses are removed for clarity and the existing operand forwarding buses (of which there are two bus spans shown) are labeled individually. The two operand forwarding (transfer) bus spans are labeled `Fa` and `Fb`. Note that each of these is generally a parallel set of buses, but for our present purposes, that distinction of not important. The single

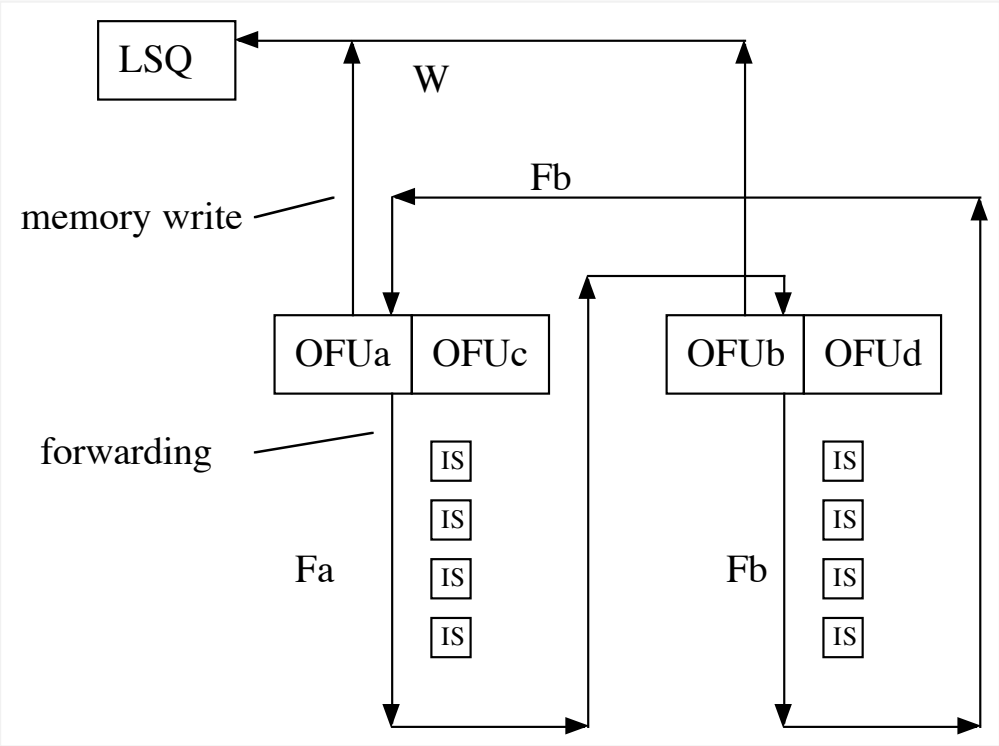

Figure 10.4: *A simplification of the previous example Levo microarchitecture.* This is a simplification of the previous figure and now also labels individual buses and operand filter units (OFU).

memory bus connecting the the operand filter units to the load-store-queue (LSQ) is also shown and labeled `W` in the figure. Each operand filter unit (four of them) is now also labeled individually. These are labeled `OFUa` through `OFUd`. Operand filter units `OFUa` and `OFUc` are physically next to each other in the silicon while the same is true of OFUs `OFUb` and `OFUd`.

Having introduced Figure 10.4, we now rearrange it logically to better visualize the machine as it commits columns. This rearrangement of the elements of Figure 10.4 is shown in Figure 10.5. In this figure (Figure 10.5), the two columns have been unfolded and shown one atop of the other in a single tall column. This arrangement is only a logical one. The same physical arrangement of components exist as they were in Figure 10.4 except they are now shown logically. All labels on the OFUs and the operand forwarding buses (still labeled `Fa` and `Fb`) represent a specific physical structure that is invariant with any logical renaming. Specifically, OFUs labeled `OFUa` and `OFUc` are still physically collocated in the layout of the silicon. Each OFU and IS are now labeled with their associated time-tags (TT) values. The time-tag value for both ISes and OFUs serve as the logical naming for those components. As the time-tags show, the oldest instructions are at the top

of the single column and the youngest (most recently dispatched) are at the bottom. Note that the time-tag (TT) of any operand filter unit is the same as the issue station (IS) it immediately precedes. The last OFU in the stacked column (`OFUc`) has the time-tag value (being eight in the figure) that the next column of ISes will have once a machine shift occurs. Everything else in the figure is otherwise the same as that of Figure 10.4. We are now in a position to describe column commitment and column shift in additional detail.

Using Figure 10.5 as our reference, our example starts with the committed architected register state being stored in the register filtering unit (RFU) residing with the overall operand filtering unit (OFU) labeled `OFUa`. Committed memory write operands may also be stored within the memory filtering unit (MFU) with this same OFU (`OFUa`). The committed predicate state consists of only a fall-through predicate (`Pf` – see Chapter 7) normally having the value of TRUE and is stored within the predicate filtering unit (PFU) of `OFUa` also. The OFUs labeled `OFUb` and `OFUc` contain speculative state only at this point in time. The remaining OFU (`OFUd`) is presently unused. How `OFUd` comes into play will be seen later. Although all of the persistent and committed architected register state is located within some RFU (within `OFUa` in Figure 10.5), neither all nor any of the persistent and committed architected memory state must reside within an MFU (within an OFU) at any given time. This is because the MFUs really only contain a cache of the whole architected memory space and may only have a few valid (or no) entries at any given time. As execution and re-execution proceeds from the ISes, new speculative operands of all types (register, memory, and predicate) start to accumulate in `OFUb` and `OFUc`. Simultaneously with instruction executions in the ISes, all operands from `OFUa` get transferred to `OFUb`. These transfers can happen either using the operand transfer buses (labeled `Fa`) or through a separate transfer interconnect for that purpose. The exact mechanism of transfer is a design choice for a particular machine. In the present example, we assume that these transfers occur over the operand transfer buses labeled `Fa`. Note that the transfer of operands from `OFUa` to `OFUb` using the operand transfer buses `Fa` also allows for intervening ISes that require input operands to snoop for them. Retirement of instructions (either commitment or abandonment) proceeds from the IS with a time-tag value of zero, but individual ISes are not allowed to retire independent of the column that they are a part of. Rather, an entire column of ISes must retire as a group. If some ISes (or perhaps even all of them) of a column are not enabled for execution or are otherwise not the correct output path of a control-flow change instruction, those ISes simply get abandoned, otherwise they get committed (their output state becomes part of the committed state of the program). Retirement of an IS is determined when all preceding ISes are ready for retirement and all output operands from a candidate IS have been forwarded. This condition cascades until the whole of a column is ready for retirement. At the point in which an entire column is ready to be retired (and committed as appropriate), all operands to be committed (of all types) will have accumulated in `OFUb` at the bottom of the top-most logical

column. At this point, both the OFU `OFUa` and the ISes of the top-most column (with time-tags valued from zero to three) can be abandoned and reassigned for new instructions. The architected register state is now stored in `OFUb` and any architected memory operands are present within the MFU of that OFU also. However, these memory operands need to be destaged back to the main memory hierarchy. This destaging is done from *OFUb* using the memory write bus labeled `W` in the figure. One the clock that this top-most column is retired, all of the time-tag values in the execution window are decremented by the number of ISes in a column (of which there are four in this present example). After column retirement, the logical view of the execution will look like what is shown in Figure 10.6.

Now referencing Figure 10.6, `OFUb` (which contained all of the newly committed operands) is now at the logical head of the entire execution window. The OFU that is partnered with `OFUb` (this being `OFUd` and formerly unused) is now placed into service as the youngest OFU at the tail of the execution window (now labeled with a time-tag value of eight). The OFU that was previously at the head of the execution window (`OFUa`) now becomes an unused OFU. The OFU that was previously at the tail of the execution window (`OFUc`, the partner of `OFUa`) is now logically located in the middle of the execution window. Also, note the interchange of the operand forwarding buses `Fa` and `Fb` exchanging places in the logical order of the execution window. We also point out that the destaging of committed memory operands that must be written back to main memory may not generally be done in a single clock period (not do they have to). Although the machine shift occurs in a single clock period, it is not necessary that memory write operand de-stage in that same period. Rather, even after `OFUb` assumes the place at the logical head of the execution window, its committed memory write operands can continue to de-stage to the LSQ. Obviously a subsequent machine shift cannot occur until all memory write operands are destaged (written back to the memory hierarchy). No physical rearrangements have occurred from Figure 10.5 to Figure 10.6, only logical renaming. These logical changes along with the retirement of a column of ISes constitutes a machine shift (being entirely logical in nature).

With the next machine shift, `OFUc` of Figure 10.6 presently located in the middle of the logical execution window will become the head, while `OFUa` will then be pressed back into service at the tail of the logical execution window. Table 10.1 shows how the four OFUs of this example machine logically rotate through successive machine shifts. The row labeled `mem-write` gives the physical OFU that may contain memory operands that need to be written back on the next machine shift transition. In the following subsections, we briefly review how each type of operand state (register, memory, and predicate) is handled during the shift operation.

Table 10.1: *Logical rotation of operand filter units on machine shifts.* Presented are logical assignments of physical operand filter units (OFUs) as the execution window undergoes successive machine shifts. Starting with the execution window in shift number zero, four subsequent machine shifts are shown.

shift state	0	1	2	3	4
TT=0	a	b	c	d	a
TT=4	b	c	d	a	b
TT=8	c	d	a	b	c
unused	d	a	b	c	d
mem-write	b	c	d	a	b

10.4.1 Register state

As already alluded to, no specific physical structure in the microarchitecture contains the persistent architected register state of the machine. Instead, the persistent architected register state is transferred among the RFUs (within the OFUs) as the machine carries out logical machine-shift operations. This feature of the microarchitecture is both novel and also serves to reduce the need for physically accessing a centralized resource. This facilitates the distributed physical scalability of this Levo microarchitecture.

10.4.2 Memory state

Persistent architected memory state shares some similarities with the persistent architected register state, except that there is too much of it (the whole of the physical address space of the main memory) to be fully contained within an OFU (or an MFU in this particulate case) like the architected registers are. Rather, since the MFUs only act as a collection mechanism for committed write operands and as a cache, they must continually de-stage their committed writes back to the memory hierarchy. Destaging occurs from the youngest MFU within the execution window when it has committed memory write operands pending. All pending committed memory write operands must be written back to the memory hierarchy before the MFU containing them can be abandoned (through a machine shift operation).

10.4.3 Predicate state

Predicate state in this machine is not architecturally visible. However, the state is otherwise treated very similarly to the architected registers. Unlike the memory operand state and like the register state, it is entirely contained within the execution window (no need for any destaging to any other backing store). However, with the predicate state, only the speculative state need be rotated around

the execution window on machine shifts. Unlike the architected state (both registers and memory), there is no committed state that needs to be accumulated for long-term persistent storage. The reason for this is that there is really only one predicate value that has any flavor of persistence throughout program execution. This can be thought of as a single architected fall-through predicate that is always TRUE. This idea is derived from the fact that program execution is always (from an architectural perspective) predicated to execute. in other words, instructions dispatched into the execution window are always expected to make progress in executing unless they are prevented from doing do so due to a control-flow change (jump) to an instruction outside of the execution window (a conditional checked for in addition to the state of the current fall-through predicate operand). Although predicate operands get accumulated in PFUs similarly to what happened with the other operand types, when they get are in a PFU that becomes the youngest PFU due to machine shifts, they get abandoned due to control-flow instruction resolutions rather than being retained as continuing persistent state. This is exactly what would be expected when one considers that the resolution of control-flow-change instructions reduce the number of outstanding execution path possibilities rather than maintaining or increasing the number of them. The introduction of control-flow-change instructions into the execution window gives rise to the need for more predicate operands while the elimination of these instructions (through either abandonment or commitment) reduces these operands.

10.5 Other operational features

In the following subsections we discuss and clarify a few of the other operational features of the machine that have not already been discussed. We briefly discuss the fetching heuristic for the machine, how persistent architected state is maintained, and how we manage multipath execution.

10.5.1 Precise interrupts

This microarchitecture also implements precise exceptions [129] similarly to how they are handled in most speculative machines. A speculative exception (whether on the mainline path or a DEE path) is held pending (not signaled in the ISA) in the IS that contains the generating instruction until it would be committed. No action is needed for pending exceptions in ISes that eventually get abandoned (squashed). When an IS with a pending exception does commit, the machine directs the architected control flow to an exception handler through the defined exception behavior for the given ISA. Interrupts can be handled in more flexible ways than exceptions. One way to handle interrupts is to allow all instructions currently being executed within the execution window to reach commitment, then architected program flow can vector off to a code handler, similarly as the case

of instruction exceptions above.

Once the control-flow of the machine has been redirected to another target instruction location, the handling of the interrupt or the exception is done exactly as it would be on a conventional machine. Specially, once the microarchitecture of the machine has redirected the instruction control-flow the remainder of any activity is simply continued architectural execution as far as the microarchitecture is concerned. This is exactly the same as what occurs on conventional machines. A software handler may perform any activities that can be done with any other software executing on the machine. Some functions of the interrupt or exception handler would generally include saving architected ISA registers to memory and also saving the instruction return address (from before the exception or interrupt) to either an ISA architected register or to memory.

It is useful to bear in mind that once the microarchitecture redirects the instruction control-flow as per the requirements of the particular ISA being implemented, all other activities in interrupt and exception handlers are identical as that which would occur on any other machine implementing the same ISA. In fact, this must be case in order for the microarchitecture to properly be ISA independent.

10.5.2 Fetch heuristics

If a conditional backward branch is predicted taken, the I-fetch unit will speculatively follow it and continue dispatching instructions into the execution window for the mainline path from the target of the branch. For a backward branch that is predicted not-taken, instructions following the not-taken output path are continued to be dispatched. If a forward branch has a near target such that it and its originating branch instruction will both fit within the execution window at the same time, then instructions following the not-taken output path of the branch are dispatched, whether or not that is the predicted path. This represents the fetching of instructions in the memory or *static* order rather than the program dynamic order. The fetching and dispatching of instructions following the not-taken output path (static program order) of a conditional branch is also advantageous for capturing hammock styled branch constructs. Since simple single-sided hammock branches generally have near targets, they are captured within the execution window and control flow is automatically managed by the dynamic instruction predicates. This is a more efficient way to handle simple single-sided hammocks than other some other methods, like that by Sankaranarayanan and Skadron with their Skipper microarchitecture. [116]

10.5.3 Multipath execution

DEE paths are created by dispatching instructions to a free column of ISes that is designated for holding DEE paths. Each SG column contains two columns of issue stations. One column is

reserved for executing the mainline path of the program, while the second column is reserved for a possible DEE path. It is not necessary that a DEE path be spawned for all mainline path segments within an SG.

All alternative paths are distinguished within the machine by a path ID value greater than one (mainline paths are always designated with a path ID value of zero). A variety of strategies can be employed for determining when spawning a DEE path is possibly advantageous for performance reasons, but generally it is when the outcome of a forward branch is either weakly predicted or if the target of the branch is both close in numbers of instructions to the branch and current DEE resources (a free column of DEE designated ISes) are abundant (can be possible wasted). The spawning of DEE paths on the occurrence of backward conditional branches introduces some added complexity and has not been well explored.

10.6 Simulation and experimental results

In this section we perform some simulations of our Levo microarchitecture in order to get a first approximation on its performance. We first present the methodology used for these simulations followed by various results. We have two goals for our simulations. First, we want to provide a comparison of the Levo microarchitecture against a conventional superscalar microarchitecture. And second, we want to explore the performance of some design-point variations of the Levo microarchitecture itself. This second goal is explored by both varying the total numbers of the Levo machine components as compared with over a baseline Levo machine and also by varying the configuration (geometry) in a few different ways.

10.6.1 Simulation methodology

Since we want to compare the Levo microarchitecture with that of a conventional superscalar machine, we really need two simulators. One will perform functional simulation for the Levo microarchitecture and the other for the conventional superscalar. Although many details of these two machines are the same (like their ISAs and most of their memory subsystems), a different functional simulator is needed for each since they are so very different in how their respective execution windows are organized.

The conventional superscalar machine that we will use as a baseline comparison for Levo is essentially the Silicon Graphics Incorporated (SGI) MIPS R10000 microarchitecture. This machine is very representative of most existing (conventional) superscalar machines of today. Although the organization of the two microarchitectures to be compared are very different, a comparison is

nonetheless possible since we can configure each machine with similar numbers of functional components. For example, each machine can be configured with the identical function-execution units, fetch widths, branch predictors, memory hierarchy, numbers of instruction reservation stations (however they are manifested in the respective microarchitecture), and a variety of other elements. The common elements of both machines is discussed in a following section. The following two subsections will discuss the respective simulator that will be used for each of the two machines that we want to compare.

Simulator for the Levo microarchitecture

Due to both the operational complexity of the individual machine hardware components and the shear size of the machines that we want to explore (in numbers of various components), the simulation methodology of Chapter 5 is not able to be employed due to unreasonable simulation execution times (at least not at the present). Rather, for these simulations we employ a reduced featured simulator that can both capture most of the functional essence of the Levo microarchitecture while simultaneously executing very quickly. This simulator is called *FastLevo*, and it was specifically designed with the above goals in mind, being very specific to the Levo microarchitecture itself. Further information on the FastLevo simulator can be found in the work of Khalafi [69].

Simulator for the conventional microarchitecture

We will employ one of the most popular simulators for characterizing our conventional superscalar baseline machine. This will be the Simplescalar MASE simulator that is distributed with the Simplescalar simulation framework. [6] This simulator is widely used for explorations of existing superscalar machines and mostly closely resembles the SGI MIPS R10000, as we have already noted above as being very representative of most conventional superscalars.

10.6.2 Experimental setup

In order to compare the two machines, we use a set of program benchmarks that represent the types of work flow that we want to target. We chose to target a work load that is both serial and sequential (not obviously parallelizable at all at the program level) in nature as well as relatively general purpose. We have chosen benchmark programs from the SpecINT suite of programs that are characterized by their sequential and integer processing nature – the type of workload that we want to address. We have chosen to use some benchmarks from both the SpecINT-1995 suite as well as from the SpecINT-2000 suite The benchmark programs used from the SpecINT-1995 suite are: COMPRESS, GO, and IJPEG. The benchmark programs used from the SpecINT-2000 suite are: BZIP2, CRAFTY, GCC,GZIP, MCF, PARSER, and VORTEX. We have used 10 benchmark

programs total. All of these programs are written in the C programming language. These programs were simulated with each of the functional simulators mentioned in the preceding sections. For use with the Simplescalar MASE simulator, the programs were compiled to the PISA instruction set (essentially identical to the MIPS instruction set but without branch delay slots) using the PISA GCC compiler. For use with our own FastLevo simulator, the programs were compiled using an SGI compiler and compiled into the MIPS instruction set. Both compilations used the standard optimization ('-O') compilation switch.

All benchmark programs used the associated SpecINT reference input files corresponding to each benchmark. For all simulations, the initialization phases of all programs are skipped using a fast-forward mechanism. Simulations were carried out by first fast-forwarding the simulated execution for 100 million instructions. This was then followed by the detailed functional simulation of the next 500 million instructions.

For all simulations (both of the conventional superscalar machine and for our Levo machine), a baseline machine configuration was adopted. The baseline machine uses a separate instruction and data L1 caches, but a unified L2 cache. The data caches use a write-back policy with a least recently used block replacement algorithm. The configuration parameters of the machine for the simulations are shown in Table 10.2. The default parameters given in Table 10.2 apply for both the baseline conventional superscalar machine as well as for our Levo machine. Additional default parameters that apply specifically to our Levo machine are given in Table 10.3. in the table the abbreviation `M-path` represents main-line (predicted) path execution while the abbreviation `D-path` represents DEE path execution (see Chapter 2). The bus span is given in terms of the number of sharing groups that share a bus before encountering the next operand filter unit.

10.6.3 Results

In this section we present variety of experimental simulation results of the Levo machine and the baseline conventional superscalar machine. We first present a comparison of the Levo machine performance against that of the baseline machine with the same hardware resources. We then explore how the Levo machine performance changes with some different machine geometries. And finally, we explore the Levo machine performance with relaxed restraining parameters for: memory latency, branch prediction, and finally both combined. The performance metric we use for the evaluation (or comparison) of the machines is the instructions per clock (IPC) a machine achieves during its execution of the benchmark programs.

Table 10.2: *Common general machine characteristics.* These machine parameters are used for all simulations for both the baseline conventional superscalar machine as well as for the Levo machine, unless otherwise specified.

parameter	value
machine word size	32 bits
processing element latencies and pipelining	same as for SGI R4000
L1 I cache size	64 KBytes
L1 I organization	2-way set associative
L1 I block size	32 bytes
L1 I cache access latency	1 clock
L1 I cache bus access delay	1 clock
L1 D cache size	64 KBytes
L1 D organization	2-way set associative
L1 D block size	32 bytes
L1 D cache access latency	2 clocks
L1 D cache bus access delay	1 clock
L2 cache size	2 MBytes
L2 organization	direct mapped
L2 block size	32 bytes
L2 cache access latency	20 clocks
main memory access latency	100 clocks
branch predictor	2-level w/ XOR
	4096 global PHT entries
	8 history bits
	1024 BHT entries
	sat. 2-bit counter
return address stack	2 each w/ 16 entries

Table 10.3: *Machine characteristics peculiar to the Levo microarchitecture.* These machine parameters are additionally used as the default for the Levo machine simulations.

parameter	value
machine word size	32 bits
main memory interleave factor	4
bus span	8
spanning bus delay w/ no contention	1 cycle
forwarding unit delay	1 cycle
parallel buses per RFU and MFU	2 (input and output)
buses per PFU	1 (input and output)
M-path to D-path column switch delay	1 cycle
D-path columns	1 per M-path column

Comparison of baseline and Levo machines

Here we present a performance comparison of the baseline superscalar machine (essentially an SGI R10000) with our Levo machine. For this comparison, each machine is configured identically in terms of its hardware resources as shown in Table 10.2. The parameters in that table specify the defaults. Additionally, each machine is configured with the equivalent of 256 reservation stations (as realized in each). The Levo machine also performs DEE path (D-path) speculative execution. and additionally has the equivalent of 256 reservations to serve for handling DEE path execution. The conventional machine only performs what is termed main-line path (M-path) speculative execution in the Levo machine. This is speculative execution that proceeds to follow the most predicted program path (determined by the branch predictor), and is now routine among conventional superscalars. Since the Levo machine can be configured in a variety of different geometries (organization of their constituent major components), we have chosen one to be a sort of baseline representative Levo machine. This is a Levo machine with 8 columns, 8 sharing groups per column, and 4 ISes per each sharing group. Each sharing group contains one processing element that performs actual instruction execution and which must be shared among all of the ISes of a sharing group. The Levo configuration chosen has 64 such processing elements and is essentially equivalent to an issue width of 64 when equated with the conventional superscalar. This machine is represented by the geometry 3-tuple `8-4-8`. Results showing the IPC performance of each machine is given in Figure 10.7. Performance in IPC is given for each of our 10 programs along with a harmonic mean (H-MEAN) calculated across all programs. The harmonic mean is more representative of typical machine performance given a workload mix than the arithmetic mean we we use it as our general machine performance characterization. The IPCs of the baseline superscalar are shown using the

bars labeled `base` while those of the Levo machine are given with those labeled by its geometry 3-tuple `8-4-8`.

From Figure 10.7, we note that the conventional machine performs very well achieving a harmonic mean of 1.96. This is very good performance that is not typical of conventional superscalars. But our baseline superscalar machine contains a large number of resources (reservation stations and execution units) that represents the upper envelope of what is typically implemented today. But even with the very good performance of the conventional superscalar machine, the performance of the Levo machine outperforms it for each benchmark. The Levo machine almost achieves a 100% performance gain over that of the conventional baseline machine in most benchmarks. It achieves a harmonic mean IPC of 3.96, which is 103% better than that of the conventional baseline machine. Further, we note that the Levo machine tends to track the performance (although scaled up) of the conventional baseline machine for each benchmark. This is not unexpected as each benchmark program presents unique challenges to both the memory hierarchy (main memory latencies and cache latencies) and to the branch predictor, all of which are the same in each machine.

Characterization of Levo machine geometries

In this section we attempt to characterize the performance of the Levo machine while changing some of its geometrical dimensions. We will perform experiments where we change one of the three basic geometrical dimensions of the machine while holding the other two dimensions constant. The geometrical dimensions that we will be varying in turn are:

- the number of columns of the machine
- the number of sharing groups in a column
- the number of issue stations per sharing group

The resulting performance of each of the configured machines is measured in instructions per clock (IPC). Results for all benchmark programs are shown along with the harmonic mean IPC across all benchmarks. The following subsections show the results of each of these explorations.

Something useful to keep in mind while examining the IPC results is that the effective issue width of any machine geometry is the number of sharing groups per column (SGs/col, the first member of the geometry 3-tuple) multiplied by the number of columns (cols, the third member of the geometry 3-tuple). It might be expected that machine performance should simply scale with the effective issue width of a machine, but this is not generally the case. The total number of issue stations is the product of all three members of the machine geometry tuple. Likewise with the effective machine issue width, performance does not necessarily track the total number of issue stations. Rather, as is

shown by the following experiments, machine performance is some (chaotic?) mixture of numbers of machine components represented by the three members of machine geometry (3-tuple) along with other effects caused by the organization of the machine itself in quantizing column retirement (due to column length) and creating different amounts of operand transfer latencies from varying numbers of operand filter units (OFUs) being traversed due to how the constant operand transfer bus span of eight (for all machine geometries explored with the exception of the `4-4-8` geometry, which has an operand transfer bus span of four). interacts with the column lengths. However, in spite of the complicated nature of the interaction of the machine operation with its associated geometry, some conclusions can still be made after examination of all of the results.

Figure 10.8 shows the IPC results of varying the number of columns for a basic Levo machine while maintaining the number of sharing groups per column at eight and the number of issue stations per sharing group at four. The number of columns varied ranges from a low of four to a high of sixteen, incremented successively by four columns for each of four geometries in this group. Although our Levo machine used for comparison with the baseline superscalar was that of the `8-4-8` geometry, for this experiment we use a smaller sized machine with a geometry of `8-4-4` to serve as a base for exploring adding columns (although the `8-4-8` geometry is also included in the results). These results show that increasing the size of the machine by adding columns (but having everything else the same) does improve performance, but only marginally. Adding columns to the machine has the effect of both increasing the window size (with the addition of more issue stations) but it also lengthens the time for operands to traverse from the top of the machine (the oldest instructions) to the bottom of it (where the youngest instructions are). The addition of each column adds a mandatory set of operand filter units (OFUs) that must be traversed by forwarded operands. As previously discussed, each column requires at least one set of operand filter units (at the top of the column) in order to maintain persistent machine state as the machine logically rotates columns through their renaming. Since each column of the machine is identically configured (numbers of sharing groups, execution units, and number of issue stations per sharing group), the addition of a column represents a corresponding increase in the total number of components for the machine within the execution window. Also, since each column is the same as another, the operand transfer bus spans remain constant for all configurations explored in this experiment. The doubling of the number of columns (for example from four to eight and then to sixteen) therefore represents a respective doubling of the total number of machine components within the execution window.

The `8-4-4` machine yields an harmonic mean IPC of 3.66. The next larger sized machine (the `8-4-8`) yields an harmonic mean of 3.98. Although this latter machine yields an 8.7% improved performance, it only does so at the expense of doubling the total number of components within the execution window. The still larger sized machines (`8-4-12` and `8-4-16`) yield more performance

(harmonic mean speedups of 13.4% and 15.6% respectively) over the base `8-4-4` machine, but with even more diminishing returns with the `8-4-16` machine having quadruple the number of machine components as the base `8-4-4` machine has (since it has four times the number of columns). Clearly, adding columns alone to the Levo machine is not necessarily the best use of additional hardware. Some other variation on the Levo machine geometry is likely to be a better investment for additional hardware. Varying the number of issue stations per sharing groups while maintaining everything else constant is considered next.

Figure 10.9 shows the IPC results of varying the number of issue stations per sharing group for a basic Levo machine while maintaining the number of sharing groups per column at eight and the total number of columns at eight. The number of issue stations per sharing groups are varied from a low of four to a high of sixteen in successive increments of four, for a total of four machine geometries in this group. For this experiment we again use our standard base machine geometry of `8-4-8` for comparison with larger sized Levo machines. As expected, the IPC performance of the Levo machine does increase with increasing numbers of issue stations per sharing group, but again with diminishing margin returns. Diminishing marginal returns are expected since increasing the numbers of issue stations per sharing group only increases the overall execution window size while not increasing the number of execution units available for actual instruction execution (computation). With this experiment we are increasing the overall execution window size but not with the burden of increasing the numbers of operand filter units that must be traversed as with the previous experiment (increasing numbers of columns). In this experiment both the operand transfer bus spans (being eight) and the number of operand filter units per column remains constant since bus spans are measured in numbers of sharing groups traversed per physical bus (which remain constant in this example) and are not measured in the total number of issue stations per physical bus. The increase in the overall execution window size does allow for increased exploitation of instruction level parallelism by finding additional opportunities for parallel execution of instruction dependency chains but the fact that the overall issue width (number of parallel executions) remains constant ultimately limits the effectiveness of the window size enlargement.

Unlike with the previous experiment, the total machine resources are not being increased uniformly with the addition of four issue stations per sharing group for each larger sized machine explored. Since the silicon resources making up an issue station is the smallest of the three main machine components within the execution window (these being: issue stations, operand filter units, and execution units), the increase of four issue stations at each larger sized machine explored is very modest compared with adding entire columns (as was first done above). Scanning the results of Figure 10.9 also shows better incremental IPC performance than those obtained with incrementally added columns. Using the machine geometry of `8-4-8` as a reference point (the base machine for this experiment) and comparing it to the next larger machines, speedups of 17.8%, 24.1%, and

24.6% are achieved respectively for the increase in issue stations per sharing group by a factor of 2, 3, and 4. These are substantially larger speedups than what was obtained by increasing the number of columns over the same `8-4-8` machine in the previous experiment (shown in Figure 10.8). These results show that adding issue stations to sharing groups (where all issue stations in a sharing group share the same instruction execution resource) is a much better use of additional hardware than using that same hardware for additional columns. But the benefit has a clear limit. We note especially that the `8-16-8` machine does not perform appreciably better than the smaller sized `8-12-8` machine. This result means that competition for the shared execution resource within each sharing group starts to degrade overall performance gains (severely) after already configuring twelve issue stations per sharing group. We next explore the effects of increasing the number of sharing groups per column while keeping everything else the same.

Now we show the IPC results of varying the number of sharing groups per column for two different base Levo machines. The first of these base Levo machines to be explored holds the number of issue stations per sharing group constant at four and the total number of columns constant at eight, while varying the number of sharing groups per column from four to sixteen, in successive increments of four, for a total of four machine geometries in this group. The smallest of these machines therefore has the geometry of `4-4-8`. The IPC results for these machine geometries are shown in Figure 10.10. As expected in exploring this machine group (results shown in Figure 10.10), performance increases with increasing machine size. But a fairly clear distinction can still be made in the way that performance increases with increasing sharing groups per column. When increasing the number of sharing groups per column (by four) from the machine geometry of `4-4-8` to the next larger sized machine with the geometry of `8-4-8`, there is an increase in IPC from 2.71 to 3.98 (an increase of 1.27, or speedup of 47%). However, increasing the number of sharing groups per column again by four to the `12-4-8` geometry (with an IPC of 4.25), and then again to the `16-4-8` geometry (with an IPC of 4.31), the IPC only increments from the previous machine by 0.27 and 0.06 respectively. These represent incremental speedups over the previous sized machines of 6.8% and 1.4% respectively. These latter performance improvements are much less than might even be expected with some constant diminishing margin return over the whole range of machine geometries. The outlying machine here in this group is the smallest, that of the `4-4-8` geometry. It appears to have a lower than expected poor performance compared with the general group. This can be explained from the fact that it has an operand bus span of four rather than a span of eight as all of the rest of the machines have. The short operand transfer bus span means that more operand filter units are traversed per sharing group than with the other machines in this group. This indicates that the operand transfer latency is not an insignificant factor in the total machine performance. This conclusion will be seen later some some subsequent machine comparisons also.

Our second base Levo machine to be explored by varying the number of sharing groups per column (SGs/col) is larger than the first and consists of holding the number of issue stations per sharing group at eight (double the size of the first baseline machine) and the total number of columns at eight (the same as with the previous set of machine geometries), while varying the number of sharing group per column from eight to 32. Each successive machine geometry has its number of sharing groups per column doubled over the previous geometry. A total of three machine geometries is explored in this group. The smallest of these machines in this group has the geometry of `8-8-8`. The IPC results for these machine geometries are shown in Figure 10.11. The results of this group of machine geometries continues to show diminishing marginal performance return with increasing machine size, but now to an even larger extent than that seen with the previous geometry experiments. Although the third geometry in this group (that of `32-8-8`) has the highest IPC achieved so far of 5.17, it is also the largest machine explored so far (with a total of 2048 issue stations and an effective issue width of 256). First, with this geometry group we are already starting out with a fairly large machine (the `8-8-8` geometry) as compared with the base machine of the other geometry groups already explored. Additionally, rather than increment the number of sharing groups per column by four as we did with the last geometry group, we are doubling that number with each successive geometry (from a base of eight, doubled to sixteen, and then doubled to 32). The resulting machine performances are IPCs of 4.69, 4.85, and 5.17 respectively (shown in Figure 10.11). The incremental speedups of the second and third machines over the previous ones are 3.4% and 6.6% respectively (these are speedups of 3.4% and 10.2% over the base machine in the group). These are fairly poor performance speedups when considering the numbers of additional sharing groups per column being added to each of these machines over the base geometry. An explanation for the lower than expected performance of these machine geometries than what might be expected due to their sizes is that they are required to incur an additional operand filter unit delay for each column than the base machine in this group. The base machine (`8-8-8`) has a single operand transfer bus span for each column, while the second machine (`16-8-8`) has two operand transfer bus spans, and the third machine (`32-8-8`) has four per column. This again shows that operand transfer delays (due to machine organization) play a significant role in overall machine performance in addition to the various numbers and types of machine components employed.

Miscellaneous geometry comparisons

In the prior section we explored the behavior of the Levo machine as we varied three of its geometry parameters. Some conclusions about which parameters produce better IPC performance when increased were indicated. In this section we want to attempt to make more definitive observations about which of differing types of machine size increases make for better performing machines. In

order to do this, we will examine machines that have the same numbers of issue stations but which have them organized in different configurations in order to gain additional insights about how the operation of the basic Levo machine changes due to these different configurations. We will examine the outcomes of two experiments along these lines.

In our first experiment, we take two machines geometries that each have 1024 issue stations and compare their performance. These two machines have the geometries of `16-8-8` and `8-16-8` respectively. These machine each have the same number of columns, but differ in their other two major geometrical dimensions. Although these two machines each have the same number of issue stations, they are organized in different ways. The first geometry (`16-8-8`) has an effective issue width of 128, while the second (the `8-16-8` geometry) has an effective issue width of only 64 (half that of the first machine geometry. Our initial expectation would be that the machine with the higher issue width (the first of the two) should perform better. But the operation of these machines (Levo machines) is not at all intuitive by simply counting numbers of components in the various geometrical dimensions of each machine. In Figure 10.12 we present the IPC performance results for these two machine geometries.

As can be seen from the IPC results shown in Figure 10.12 each of the two machines performs very similarly. The second machine has a slight performance advantage with its harmonic mean IPC of 4.96 while the first machine only achieved an harmonic mean IPC of 4.85. Our expectation was that the first machine should have performed better since it had an effective issue width (execution resources) of 256 while the first had only an effective issue width of 128 (half as much). But not only did our expectation fail, but the second machine (with only half the effective issue width) performed slightly better. What is going on in these machines?

The first machine had sixteen sharing groups per column and eight issue stations sharing the execution resource of each sharing group. The second machine had only eight sharing groups per column but with sixteen issue stations per sharing group, contending for use of the single execution resource in each sharing group. Again, these configurations might lead us to think that the first machine with both more execution resources and with less contention for access to the same execution resources would perform better (for each of the two reasons just stated). However this is not what happens. What is not immediately obvious is that the first machine, with its sixteen sharing groups per column, also has two operand filter units per column. The second machine, with only eight sharing groups per column, only has a single operand filter unit per column (the minimum possible for rotation of the persistent state of the machine). Both machines share a common trait of having a constant operand transfer bus span of eight (eight sharing groups traversed before encountering another operand filtering unit). It is the presence of the additional operand filter units per column on the first machine that limits the other advantages (more execution resources, and secondly, less contention for each execution resource) of the first

machine over that of the second. In fact, the first machine has exactly twice the total number of operand filter units than the second one does, even though both machines have the same overall number of issue stations. The presence of the additional operand filter units in the first machine geometry creates some additional delay in transporting operands from each column to the next. It is this additional delay in operand transport that accounts for the moderation of the performance of the first machine. This result shows that a simple expectation or trend for machine performance for any given machine geometry is not automatically discernible. The reason is that modifying one geometrical parameter of a machine at the expense of another changes more than a single operational aspect of the new machine. This general problem in predicting machine performance is illustrated in another experimental observation examined next.

In a second experimental observation, we again compare the IPC performance of two different machine geometries. In this experiment we compared machines with the same number of columns (same as with the first experiment above) and again varied the number of sharing groups per column and the number of issue stations per sharing group, but in a more dramatic way as compared with the prior experiment above. Again in this comparison, both machines have the same total number of issue stations. The two machine geometries compared are `16-4-8` and `8-8-8`. Each of these machines have a total number of issue stations of 512. The first has an effective issue width of 128 while the second has an effective issue width of only 64 (half that of the first machine). Again, the most intuitive conclusion about which machine should perform better would be that the first should outperform the second. This would be a reasonable conclusion since the first machine has double the execution resources of the second, while also incurring only half of the contention for each execution resource as compared with the second. Again, the intuitive conclusion does not hold. The IPC results for the two machine geometries are shown in Figure 10.13.

In this experimental comparison (IPC results shown in Figure 10.13), although both machine geometries share the same number of columns and total number of issue stations, the performance of the two machines is not the same. The second machine performs significantly better than the first. The first machine achieves an harmonic mean IPC of 4.31, while the second machine achieves an harmonic IPC of 4.69 (a speedup of 8.8% over the first). Similarly to the prior experimental observation above, the reason for the performance discrepancy is the same. The first machine has two operand filter units per each column, while the second has only one operand filter unit per column (the minimum required). Each machine also shares a constant operand transfer bus span of eight. The additional delay in transporting operands from each column to the next in the first machine offsets the other advantages of the first machine over the second. This result again shows that a simple expectation for machine performance is not readily attainable for machines with geometries that differ in more than the most simple way (the extension of single dimension by some amount).

The speedup of the second machine geometry in this second set of machines examined (with geometries of 16-4-8 and 8-8-8) is greater than the corresponding speedup in the first set of machines examined previously (with geometries of 16-8-8 and 8-16-8) due to the different ratios of issue stations per column to the number of operand filter units per column in each of the geometry sets. The first set of machine geometries examined had ratios of issue stations per column to the number of operand filter units per column of 64 and 128 respectively for the geometries 16-8-8 and 8-16-8. The speedup of the second machine geometry in this set over the first was 2.3%. The second set of machine geometries examined had ratios of 32 and 64 respectively for the geometries 16-4-8 and 8-8-8. The speedup of the second machine geometry in this set over the first was 8.8%. The performance speedup of the second machine (8-8-8) over the first (16-4-8) in the second set was larger because of the smaller ratios of issue stations per column to the number of operand filter units per column. The larger ratios of these parameters that exist in the first set of machines (16-8-8 and 8-16-8) tends to reduce (or to less dramatize) the effect of the differing numbers of operand filter units per column in each individual machine geometry examined as compared with those in the second set of machine geometries. The smaller ratio of issue stations per column to operand filter units per column that exists in the second set of machines (16-4-8 and 8-8-8) tends to more dramatize the effect of the differing numbers of operand filter units per column in each individual machine of that set. This is an expected result. This is expected since when more issue stations operate over the same per-unit number of operand filter units it should tend to minimize the effect of the operand transfer delay caused by those operand filter units. In effect, the operand transfer delay caused by traversing the operand filter units is amortized over a larger number of issue stations in the first set of machine geometries examined in this section, causing the individual machines to perform much more closely to each other than in the second set of geometries examined.

For a general reference, a summary of the harmonic mean IPC (across all benchmarks) for each of several machine geometries is shown in Figure 10.14. The numeric values of these harmonic mean IPCs is given in Table 10.4.

From the general harmonic mean IPC results presented in Figure 10.14 and Table 10.4 some additional comparisons and conclusions can be made. The two machine geometries of 16-4-8 and 8-8-8 have already been analyzed in detail above. Each of these machine geometries had the same number of issue stations and the same number of columns and were interesting for that reason. The 16-4-8 geometry had twice the issue width of the other machine but performed more poorly than the 8-8-8 geometry machine. But it is also interesting to compare the 8-4-16 geometry machine to each of those two previous machines. This machine (8-4-16) also has the same number of issue stations as each of the other two, but it has twice the number of columns as compared with those. Comparing the 8-4-16 geometry machine first to the 8-8-8 machine, we note that it has twice the issue width (execution units). This might lead to the conclusion that it should outperform

Table 10.4: *Harmonic mean IPC values for a variety of Levo machine geometries.* The harmonic mean IPC values for a variety of Levo machine geometries are provided for reference. Harmonic means are computed over all benchmarks for each machine geometry. Machine geometries are specified in column one of the table using the 3-tuple representation.

geometry	H-MEAN IPC
4-4-8	2.71
8-4-4	3.66
8-4-8	3.98
8-4-12	4.15
12-4-8	4.25
16-4-8	4.31
8-4-16	4.23
8-8-8	4.69
8-12-8	4.94
8-16-8	4.96
16-8-8	4.85
32-8-8	5.17

the `8-8-8` machine (due to twice the issue width), but this is obviously not the case. The `8-8-8` machine has an harmonic mean IPC of 4.69 while the `8-4-18` has only an harmonic mean IPC of 4.23. Both of these machines also have the same number of operand filter units per column (one OFU per column) so this in itself is not a difference. These machines also share the same operand transfer bus span (being eight). However, since it has twice the number of columns as compared with the `8-8-8` machine, it effectively has more operand filter units to traverse for its entire set of issue stations than does the `8-8-8` machine. This accounts for its reduced performance as compared with the `8-8-8` machine. This result shows that a machine having additional columns can perform similarly to one with simply additional operand filter units for the same total number of sharing groups (or the same total number of issue stations). This is not unexpected since the columns of a machine are just the folded representation of its total ring structure.

But a more interesting and subtle comparison is between the `8-4-16` geometry machine and the `16-4-8` geometry machine. These two machines share the same number of issue stations, operand transfer bus span, and execution resources. The difference is that the `8-4-16` machine has twice the columns, while the `16-4-8` machine has twice the number of operand filter units per column. Each have the same total number of operand filters units fot its entire execution window. We would therefore expect these two machines to perform very similarly. They do indeed perform very similarly, but the `16-4-8` machine performs slightly better. The distinguishing factor is that the

8-4-16 machine has its operand filter units organized in more columns rather than having more in each column. It must be remembered that the machine rotates and retires instructions based on the fundamental quantization of a column. This column quantization seems to make the difference. This effect is not just realized with these two machines (8-4-16 and 16-34-8). It can be noted that it also shows up with the 8-4-12 and 12-4-8 geometry machines. These latter machines have the same sort of organizational tradeoffs as compared with the 8-4-16 and 16-4-8 machines (replace 12 with 16 throughout). Those machines having the longer columns (more SGs per column) but with all other numbers and types of resources the same (and in the same proportions where possible), also have a slight performance advantage over other machines in its class with shorter columns (less SGs per column). This same results appears with the 4-4-8 and 8-4-4 machines (the latter having the better performance).

But in the case of the machine geometries of 4-4-8 and 8-4-4 there is an additional reason for the latter to have the better performance. This is due to it (the latter) having less total number of operand filter units. The latter machine (8-4-4) has a total number of operand filter units traversed over its execution of only four, while the 4-4-8 machine has twice that at eight operand filter units being traversed over its entire execution window. These two factors seem to contribute to the larger difference in harmonic mean IPC between these two machines than between the previous two machine pairs mentioned previously (reference Figure 10.14 and Table 10.4).

Although the results of this section show that any expectation for machine performance is not some simple inference from its geometrical configuration, the situation is not really a problematic one for machine designers. Any machine design would likely start with the amount of silicon space available for the design as a starting point. This is driven by technology and is usually the starting point for any new machine designs. Given this fact, a machine designer for a new Levo machine would start with a Levo machine of some geometrical configuration that roughly fills up the available silicon space as dictated by the target silicon technology generation. From this starting point, machine geometries around the starting geometry should be simulated in order to find the ones that perform the best. This is not unlike what occurs already with choices about caches sizes at the various memory hierarchy levels, or with any of several combinations of the numbers of other existing conventional machine components. In this respect, the Levo machine microarchitecture is not all that different (from a design perspective) than existing machine microarchitectures.

Relaxing machine constraints

In this section we explore the performance of the Levo machine microarchitecture but with some relaxed design constraints. The Levo microarchitecture is most similar to other conventional microarchitectures in its memory hierarchy and in the way that it performs instruction fetch prediction.

Table 10.5: *Simulation case suites involving relaxed machine parameters.* These four simulation case suites are executed on four machine geometries in order to explore the potential performance of the Levo machine microarchitecture.

case	memory	branch prediction
RM/RF	realistic	realistic
IM/RF	idealistic	realistic
RM/IF	realistic	idealistic
IM/IF	idealistic	idealistic

The primary difference between the Levo microarchitecture and that of conventional microarchitectures is in the part of the machine that we have termed the execution window. In this section we want to explore relaxing the constraints of both the memory hierarchy latency and the branch prediction miss rate in order to try to expose the performance potential of the core part (execution window) of the Levo microarchitecture itself.

We proceed in this section by relaxing two specific constraints on the Levo machine, both individually and together. These two constraints are: the L1 miss rate , and the branch prediction miss rate. More specifically, we explore artificially bringing the L1 miss rate to zero and the branch prediction miss rate to zero also. For the L1 memory case, this means that effectively all memory accesses will hit in the L1 instruction and data caches with the standard L1 hit latency of 1 clock cycle. For the branch prediction case, it means that there will be no mispredicted branches at instruction fetch time and no penalties incurred from the recovery from any mispredicted branches. We will carry out simulations of four combinations of non-relaxed and relaxed cases, on four different machine geometries, across all of our benchmarks. The first simulation case suite to be performed in that of no relaxation of either of the above two machine parameters (both will be realistic). This simulation case is subsequently denoted as `RM/RF` to represent `real-memory/real-fetch`. This case serves as the baseline for the subsequent three cases that have one or more idealized parameters. The second case suite will be with relaxed (idealized) memory but with realistic branch prediction. This simulation case is subsequently denoted as `IM/RF` to represent `ideal-memory/real-fetch`. The third case suite will be with relaxed (idealized) branch prediction and realistic memory behavior. This simulation case is subsequently denoted as `RM/IF` to represent `real-memory/ideal-fetch`. And finally we will perform a simulation case suite with both relaxed memory and relaxed branch prediction combined (both being idealized). This simulation case is subsequently denoted as `IM/IF` to represent `ideal-memory/real-fetch`. These four simulation case suites to be performed are summarized in table 10.5. The four machine geometries that are explored in this section are (given in terms of their primary 3-tuples):

Table 10.6: *Effective issue width and the total number of issue stations for four Levo machine geometries explored.* The effective instruction issue width and the total number of issue stations is shown for each of four Levo machine geometries used to explore the effects of relaxing L1 miss rate (memory latency) and the branch prediction rate (wrong path handling). Machine geometries are shown using their 3-tuples.

geometry	issue width	issue stations
8-4-8	64	256
8-8-8	64	512
16-8-8	128	1024
32-8-8	256	2048

- 8-4-8
- 8-8-8
- 16-8-8
- 32-8-8

All of these simulations are performed using the same ten benchmarks as before and with using all other machine parameters as shown in Tables 10.2 and 10.3 with the sole exception of the parameters to be selectively idealized. For additional reference, Table 10.6 provides the effective issue width and total number of issue stations for each of the four machine geometries used in our present analysis.

Figures 10.15, 10.16, 10.17, and 10.18 show the IPC performance results of each of the simulation cases. In these figures, results are grouped according to the machine geometry that the simulations were performed on. Each figure shows a separate machine geometry. For each machine geometry, the IPC results are shown successively for each of the four cases of Table 10.5 for each of the benchmark programs. An harmonic mean across all benchmark programs is also shown.

These machines are among the largest sized machines explored so therefore we expect some of the highest IPC performance from them as compared with any of the smaller sized machines previously examined. But more importantly, for our purposes, we want to see how each of the simulation cases with one or more idealized parameters used differs from the baseline case with all realistic parameters. From the IPC performance results shown in Figures 10.15 through 10.18 (one figure each for each machine geometry simulated) it is seen that all four of the Levo machine geometries explored performed similarly under each of the four simulation cases. The results show that both memory latency (originated from L1 misses) and branch mispredictions substantially constrain the performance of the Levo machine microarchitecture. This is not unexpected since the

Levo microarchitecture did not make any direct contribution towards eliminating either of those real-world constraints. Although there is no particular expectation for whether relaxed memory latency (L1 miss rate to zero) or idealized branch prediction (misprediction to zero) will perform better than the other, it was expected and seen that the case of both of those parameters being relaxed outperform the prior three cases. However, the IPC results do show that the case of idealized branch prediction alone almost always outperforms the case of idealized memory latency alone.

The increased performance of the case with the idealized memory but with realistic branch prediction (fetch) is certainly expected since all memory read accesses are now hitting entirely within the L1 caches. Of crouse, from these results, any use of larger L1 caches (or any other means) that a conventional microarchitecture might use to lower the L1 miss rate will also apply to benefiting the Levo microarchitecture. Although the Levo microarchitecture does not encompass or contain any direct means for reducing L1 miss penalties, the microarchitecture does include the additional machine components within its execution window that we have termed L0 caches. These L0 caches are actually located within each of the memory filter units (MFU), one L0 per MFU. The results show that either a basic improvement in the L1 miss latency or possibly some sort of improvement in the use or function of the L0 caches might lead to substantial overall performance gains.

With the relaxation of the branch prediction miss rate (to zero) for instruction fetching, an even greater performance increase is achieved over the baseline than with the L1 miss rate relaxation. This result shows that attention to branch misprediction is a large factor in Levo machine performance, and that it is as important for the Levo machine microarchitecture as it currently is for any other conventional microarchitecture. A positive result of this is that any improvement in branch misprediction (towards zero) for conventional microarchitectures will also benefit the Levo microarchitecture. This result also shows that there is the possibility for improvement in the way that DEE path execution is carried out. Better DEE path handling might serve to mitigate (at least in some part) a poorer than desired branch prediction capability.

Finally, as expected, the combination of both parameters being relaxed yields the highest performances for all machine geometries. This simply shows that improving both of these factors (L1 miss rate or latency and branch misprediction) still provides potential performance improvements for the Levo microarchitecture as opposed to improving just one of those factors. This means that each of these factors are somewhat independent of each other in their potential to improve performance, although the relaxation of branch misprediction improves performance much more so than the relaxation of memory latency.

In Figure 10.19 we have summarized the results of this section by only including the harmonic mean IPC performance across all benchmarks for each of the four machine geometries used and for

Table 10.7: *IPC speedups over the realistic base Levo machine for three cases of relaxed machine parameters.* The IPC speedups are shown for each of the three cases of relaxed machine parameters, for each of four machine geometries explored. Machine geometries are indicated using their 3-tuple representations. The three cases of relaxed machine parameters are indicated using the abbreviations given in column one of Table 10.5.

IPC speedup	IM/RF	RM/IF	IM/IF
8-4-8	1.23	1.84	2.49
8-8-8	1.29	2.30	3.10
16-8-8	1.32	2.94	3.89
32-8-8	1.33	3.26	4.08

each of the cases of relaxed parameters that we have investigated. The summarized harmonic mean IPC results of this figure more clearly shows that although the Levo machine performance increases with increasing machine size, it clearly does so mostly when branch misprediction in instruction fetching is relaxed (to zero) as compared with the relaxation of memory access latency (through L1 misses). This result is even more evident with increasing machine size. For completeness, it should be noted that although the latter three machine geometries used in these results increase in size with a corresponding increase in effective issue width (as well as with issue stations), the first machine geometry (`8-4-8`) shares the same effective issue width with the second geometry (`8-8-8`, both being 64. Again, Table Table 10.6 shows both effective issue width and the total number of issue stations for all four of these machine geometries. While the harmonic mean IPC for the idealized memory case (relaxed L1 miss rate to zero) increases respectively from 4.91 to 6.86 with the increasing sized machine geometries (a modest performance increase), the harmonic mean IPC for the idealized branch misprediction case increases respectively from 9.92 to 21.07 (a speedup of 2.14) over the same machine geometries, showing its greater impact on potential machine performance.

The speedups over the base realistic Levo machine for each of the three cases of relaxed parameters for each of the four machine geometries explored is given in Table 10.7. Referencing Table 10.7, it can be observed that the IPC speedups for each case of the relaxed machine parameters increase with increasing machine size. This indicates the there is more potential IPC to be gained with improvements in the actual performance of either or both of the relaxed machine parameters (L1 miss rate and instruction fetch branch misprediction) on larger sized Levo machines than with the smaller ones. This is good news since the larger sized machines already had substantially higher IPCs to start with than the smaller sized machines.

Also very evident from Figure 10.19 is the fact that actual machine IPC performance on the

realistically configured machines (no relaxation of either L1 miss rate or branch misprediction) does not appreciably increase with increased machine sizes. The harmonic mean IPC over all benchmarks only increases from 3.98 to 5.17 (a speedup of 1.30) over all four machine geometries. Comparing this with the speedup with the idealized branch misprediction case just mentioned (a speedup of 2.14), this again shows that much more performance potential from the Levo machine is available from increased machine sizes when design attention is devoted towards mitigating or otherwise managing the performance impact of the branch misprediction restraint.

Discussion

In the preceding sections, we have presented simulation results of the Levo machine under different circumstances and configurations. We performed three basic sets of experiments. These are discussed in turn.

We first compared the Levo machine performance with that of a similarly configured conventional superscalar machine. Both machines (the Levo machine and the conventional superscalar) each had the same number of basic components of each type. The Levo machine was of the `8-4-8` geometry. Specifically, they both had 256 components serving in the capacity of a reservation station. They each also had the equivalent issue width of 64, meaning that in principle they could execute up to 64 instructions simultaneously. Using the harmonic mean IPC, across all of the benchmarks used for the experiment, the Levo machine achieved a speedup of 2.03 over the baseline superscalar. Although both machines

Secondly, we investigated the performance of the Levo machine as its size was increased in each of its three primary dimensions (represented by the 3-tuple nomenclature). Although machine performance consistently increases as each dimension is individually increased, the increases are not similar or comparable for each dimension. One clear observation is that machine performance does not increase proportionally with increased numbers of components. This is a very general expectation for most systems that do not have strictly separated or independent resources that can be applied completely in parallel at the same time in order to multiple performance. Although the Levo machine has many components operating in parallel, those components are by no means operating strictly independently from each other, but rather exhibit many cross-coupling dependencies and structural machine hazards among them. The Levo machine therefore understandably exhibits diminishing margin performance returns with increasing size. One of the major factors for the inconsistent performance increases with increases in each of its primary dimensions (each of the dimensions within the 3-tuple nomenclature) is due to the fact that generally an increase in the size of a dimension of the machine introduces secondary effects that serve to change the dynamics of the machine in counterintuitive ways. One example of this that was highlighted in the detailed

discussion of the results was the fact that increasing the number of sharing groups in a column not only increases the number of processing elements available for instruction execution (increases effective issue width) but also increases the delay for operands to traverse the machine due to the general multiplication of operand filter units that must be traversed, itself an artifact of a constant operand transfer bus span. The general result is that there is no simple way to predict machine performance, even in a qualitative way, when even a single dimension is changed. However, many conventional superscalars share this sort of performance attribute, so it is not unique to the Levo machine.

Thirdly, we performed a set of experiments where we relaxed two key parameters of the Levo machine (these being the L1 miss rate and the branch prediction miss rate, each to zero) in order to attempt to gauge the potential performance of the Levo machine. The effects of the two relaxed parameters were to respectively idealize the memory hierarchy (beyond the L1 cache) and the efficiency of instruction fetching. These results indicated that although the Levo machine will improve its performance with a lower L1 miss latency (having to go to L2 or beyond), a very substantial potential performance increase is possible with better branch prediction during instruction fetch.

10.7 Summary and conclusions

We have presented a microarchitecture (that we have termed Levo) that embodies both the idea of Resource Flow execution as well as having the ability to be physically scaled. These objectives were not in conflict. We have drawn on the novel ideas presented in the earlier chapters to bring together what has been achieved and presented in this chapter. The Resource Flow execution model served as the base for the present work. The microarchitecture presented in Chapter 4 (OpTiFlow) served as a starting guideline for enhancement. The scheme for the dynamic predication of instructions within the microarchitecture itself (presented in chapter 7) was incorporated into the present microarchitecture. And the introduction and development of first operand forwarding units and then operand filter units (presented in chapter 8) were used to finally endow the present microarchitecture with the ability to be physically scalable.

The ideas and components associated with Resource Flow execution actually facilitated the introduction of physical scalability due to the fact that the required signal coupling between or among issue stations is so low. Drawing on the object-oriented metaphor, although the cohesion within issue stations is high, the required coupling between issue stations and the execution units is comparatively quite low. This feature of the Resource Flow execution model was exploited to facilitate the physical scalability of the machine as a whole. Further, the other required machine functions of operand dependency determination and instruction execution and re-execution were likewise handled in the context of the Resource Flow execution model.

Scalability was also achieved through the introduction of operand filter units (of various types). These units serve to both transfer operands from one physical segment of the distributed machine to another (an operand forwarding function) while also performing an operand filtering function where redundant operands are filtered out or prevented from being unnecessarily forwarded. The operand forwarding function limits the physical lengths of the operand transfer buses required to forward operands from one issue station to another. This ensures that realistic physical electrical constraints are satisfied even though the machine is otherwise physically scaled to larger sizes (more machine components). The operand filtering capability serves to free up bandwidth on the available operand forwarding buses, thus reducing operand forwarding contention and delays. These units also serve as operand caches of a sort for operands and this caching function can lead to reductions in the number of operand transfers on some bus segments, possibly leading to bus bandwidth savings as well as increased overall execution performance.

After introducing and describing the Levo machine microarchitecture itself, we performed a number of simulation experiments on it using a realistic set of sequential integer-oriented benchmarks from the SpecINT suite. Several observations and conclusions were possible from these simulation experiments. First, the Levo machine performs at approximately twice the performance (speedup of 2.0) of an equivalently configured conventional superscalar machine. This is a 100% improvement over existing conventional superscalars. Although this is a fairly positive result, the primary disadvantage of this is the introduction of a very novel microarchitecture, one which is very different than anything commercially used so far – and even very different even from any research-oriented microarchitectures. Next, various experiments evaluating the Levo microarchitecture as its size is increased in various ways shows two main results. First that the Levo machine exhibits diminishing marginal returns for increased sizes (performance does not scale with machine size). And secondly, that performance cannot be quantitatively predicted based on even a simple increase in a single dimension of the machine (using the basic 3-tuple machine geometry nomenclature). Finally, experiments to assess the potential performance of the Levo machine microarchitecture showed that although machine performance increases with relaxed main memory latency constraints, it much more dramatically increases with the relaxation of instruction fetch branch misprediction. This shows where future research should be directed in order to exploit the most potential from the Levo microarchitecture idea – reducing or mitigating branch misprediction rates or penalties.

The introduction of the Levo machine microarchitecture (an exploitation of the Resource Flow execution model) shows that large physical scalability of processors is possible. This allows for higher potential program performance gains on those single-threaded program codes of a highly sequential nature that are not suited towards execution on more explicitly parallel machines, such as multicore machines or other simultaneous multiple processing (SMP) machines. Although existing multicore processing machines may be a better use of available silicon when overall processing

throughput is desired, that approach does nothing to increase the performance of sequential single-threaded programs. The Levo machine microarchitecture offers one possible way to trade available silicon space for single-threaded program performance (albeit at diminishing returns). But since achieving performance gains on single-threaded and highly sequential programs codes is so difficult at the present time with any available microarchitecture, the introduction of the Levo microarchitectures represents a contribution to the state of the art for use when the market tradeoff between silicon space and program performance is warranted.

In addition to the fact that the Levo microarchitecture only achieves diminishing performance gains with increasing silicon space (requiring suitable market needs to warrant its use), an even greater obstacle to its adoption is its very substantial novelty. The Levo microarchitecture consists of a very different approach towards machine organization and execution than any previous microarchitecture. It approaches the problem of executing instructions from the perspective of allowing for rampant out-of-order execution while providing the mechanisms for dynamic dependency determination and eventual commitment. This approach is in stark contrast to conventional machines that rather carefully gauge the among of out-of-order parallel execution that is allowed at the outset of their machine approaches. This substantial novelty of the Levo microarchitecture will likely present both a barrier to commercial adoption as well as to further significant research on its improvement. But even in light of this, Levo still represents a possible way forward towards achieving substantial performance gains on single-threaded sequential programs codes that is not available with any other approach.

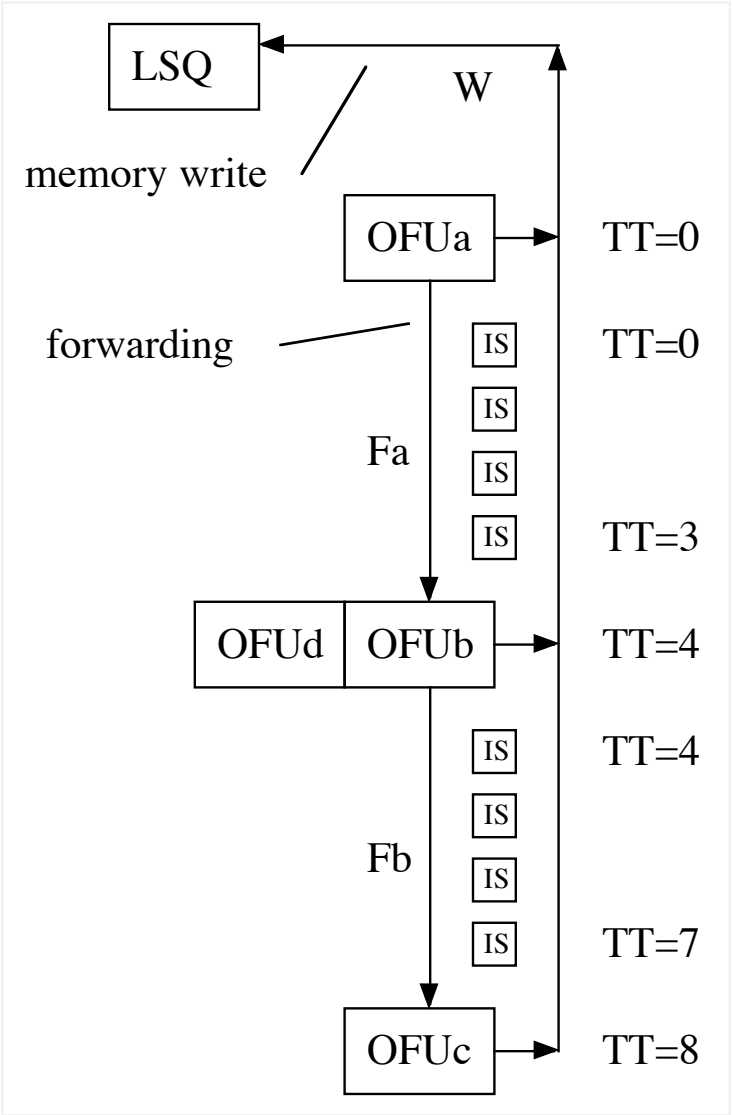

Figure 10.5: *A logical arrangement of the previous example Levo microarchitecture.* This figure shows the same example microarchitecture as Figure 10.4 but from a logical perspective. The physical columns are stacked logically so as to show the execution window of the microarchitecture.

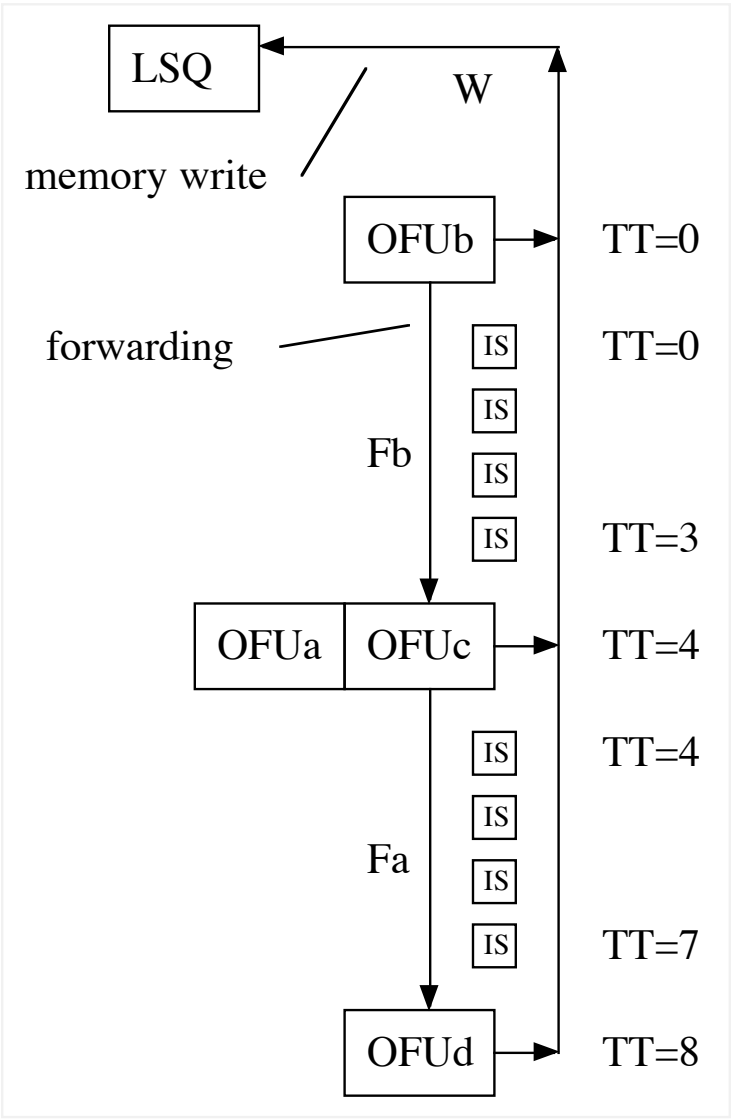

Figure 10.6: *The logical view of the execution window of the previous figure but after a machine shift.* This is the logical view of the execution window of Figure 10.5 after a machine shift has occurred.

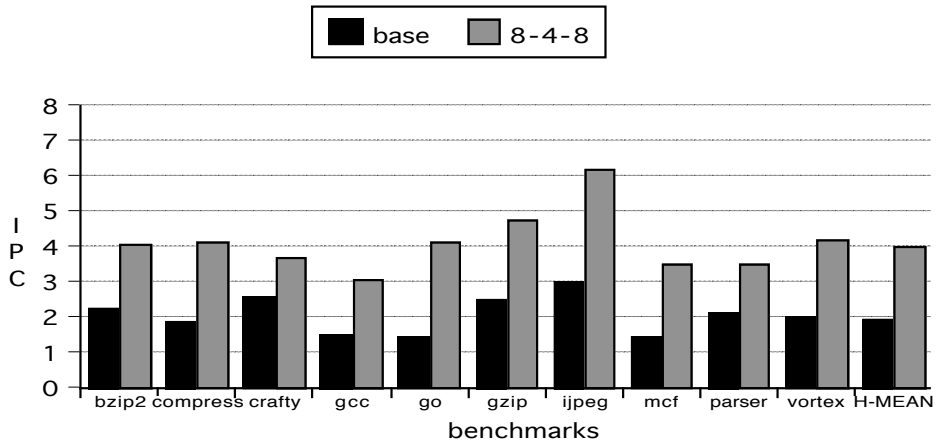

Figure 10.7: *IPC performance comparison of the conventional baseline superscalar machine with that of the Levo machine.* A performance comparison of the baseline conventional machine with that of the Levo machine is shown. Performance is measured in instructions per clock (IPC) and higher bars represent higher performance.

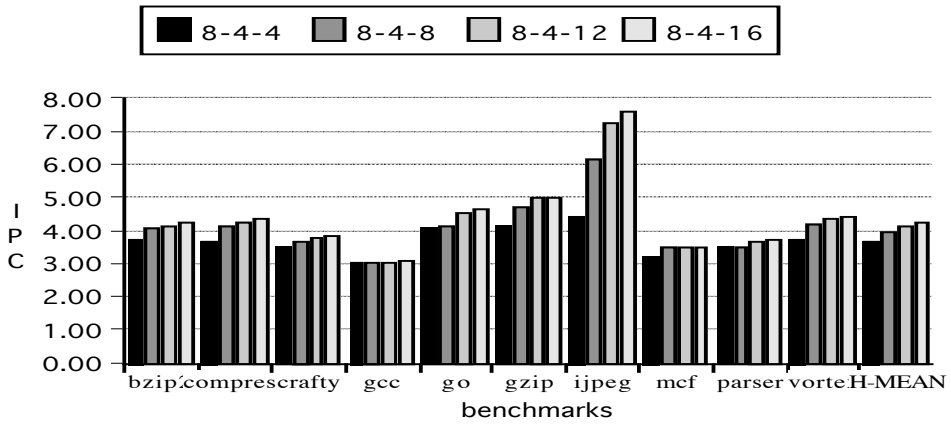

Figure 10.8: *IPC performance of a basic Levo machine but with varying columns.* The IPC performance of four Levo machines each with eight sharing groups per column and four issue stations per sharing group is shown while varying the number of columns from four to sixteen, in increments of four. The geometry notation is: SGs/col, ISes/SG, cols.

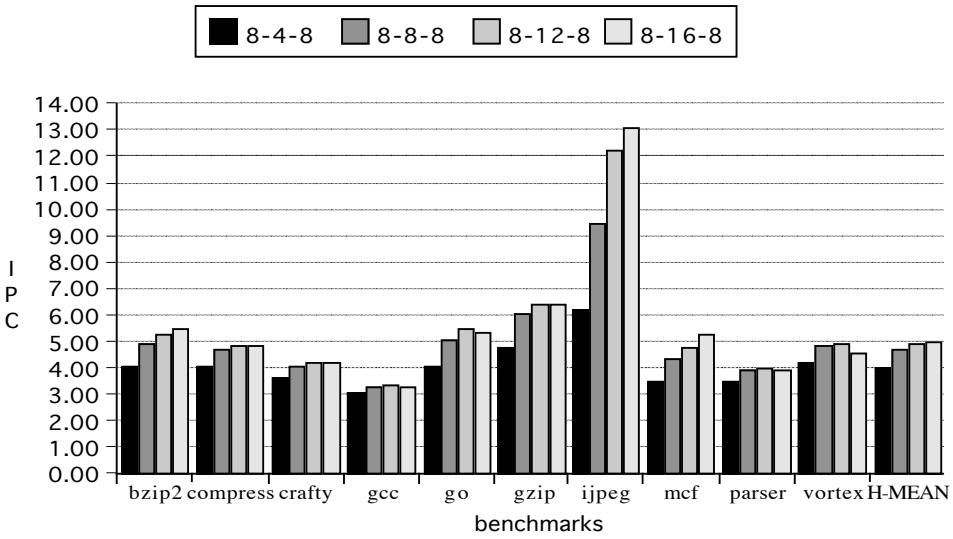

Figure 10.9: *IPC performance of a basic Levo machine with varying numbers of issue stations per sharing group.* The IPC performance of four Levo machines each with eight sharing groups per column and eight columns is shown while varying the number of issue stations per sharing group from four to sixteen, in increments of four. The geometry notation is: SGs/col, ISes/SG, cols.

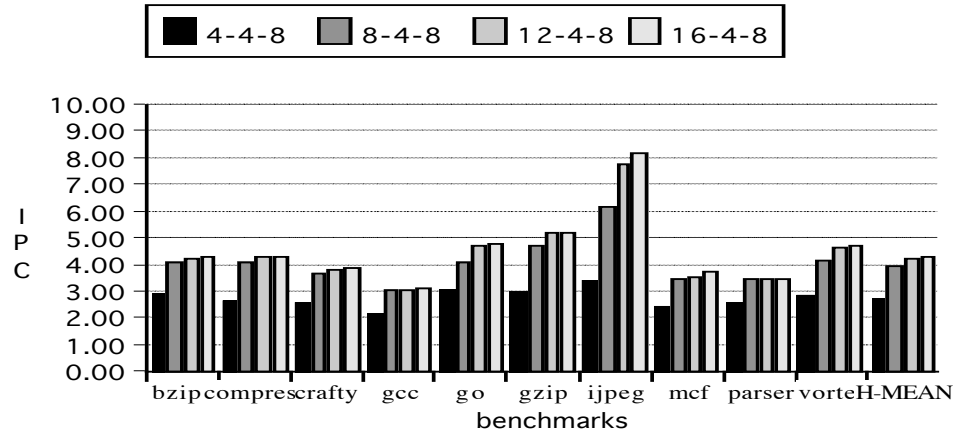

Figure 10.10: *IPC performance of a basic Levo machine with varying numbers of sharing groups per column.* The IPC performance of four Levo machines each with four issue stations per sharing group and eight columns is shown while varying the number of sharing groups per column from four to sixteen, in increments of four. The geometry notation is: SGs/col, ISes/SG, cols.

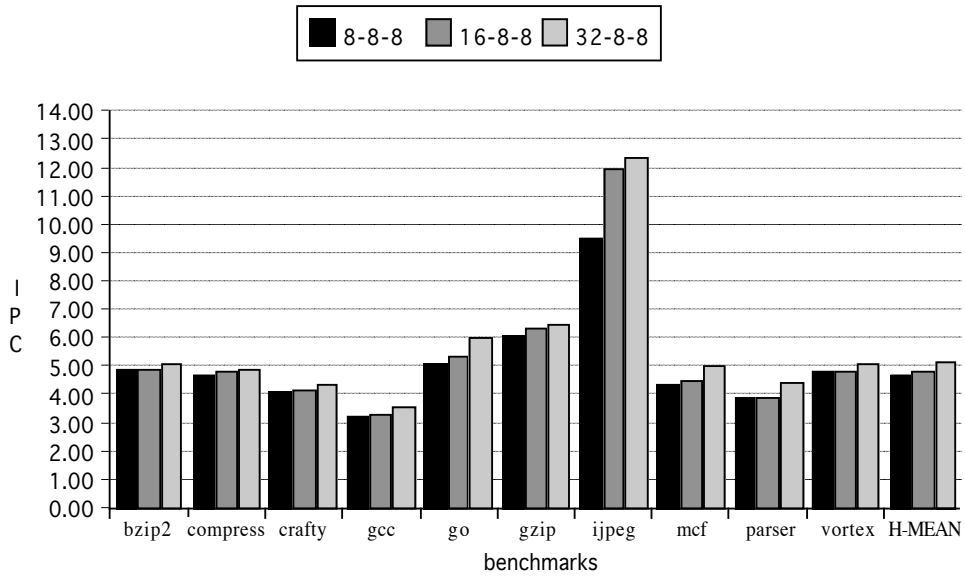

Figure 10.11: *IPC performance of a Levo machine with varying numbers of sharing groups per column.* The IPC performance of three Levo machines each with eight issue stations per sharing group and eight columns is shown while varying the number of sharing groups per column from eight to 32.

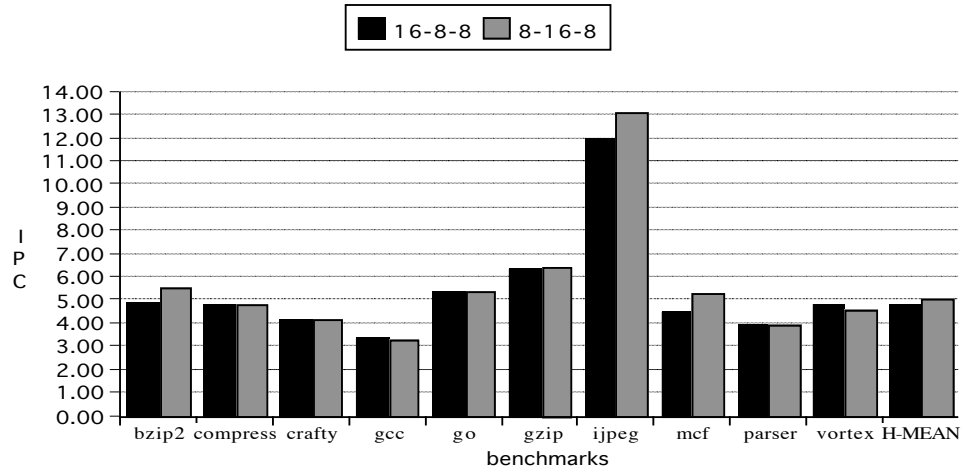

Figure 10.12: *IPC performance of two Levo machine geometries with the same total number of issue stations but with differing organizations.* The IPC performance of two different geometries of the Levo machine is shown. Each machine geometry has the same total number of issue stations and columns but with different organizations of the issue stations. The first machine has an effective issue width of 256 while the second has an effective issue width of only 128. The second machine, with a lesser issue width, performs better than the first across most of the benchmarks and therefore has a higher harmonic mean IPC than the first.

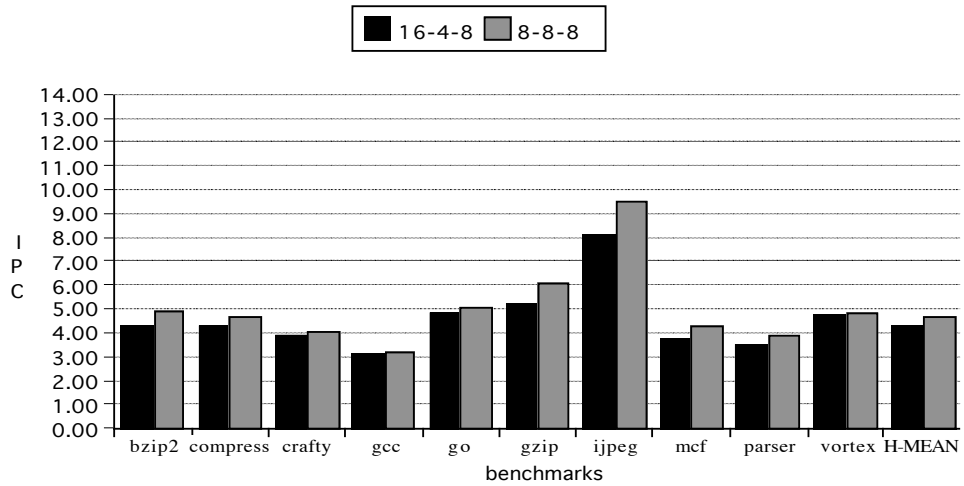

Figure 10.13: *IPC performance of two Levo machine geometries with the same total number of issue stations but with differing organizations.* The IPC performance of two different geometries of the Levo machine is shown. Each machine geometry has the same total number of issue stations and columns but with different organizations of the issue stations. The first machine has an effective issue width of 128 while the second has an effective issue width of only 64.

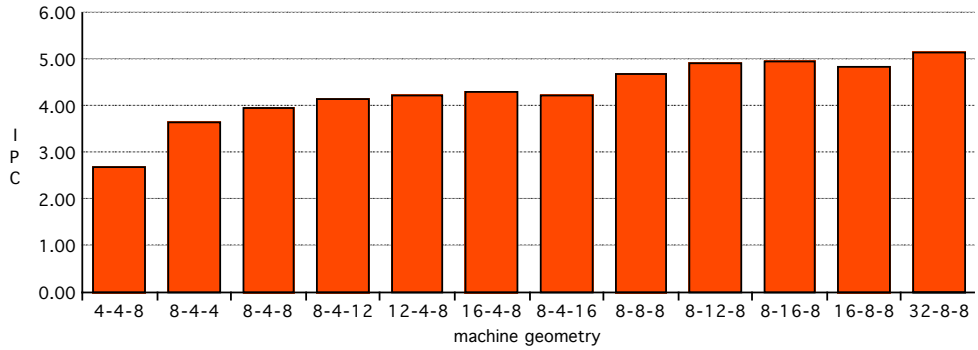

Figure 10.14: *Harmonic mean IPC performance of a variety of Levo machine geometries.* The harmonic mean IPC performance of several Levo machine geometries is shown. Machine geometries are specified using the 3-tuple representation (SGs per column, ASes per SG, and number of columns). The IPC performance does not always increase with increasing machine resources. Rather, IPC performance depends in part of some subtle features of the machine geometries.

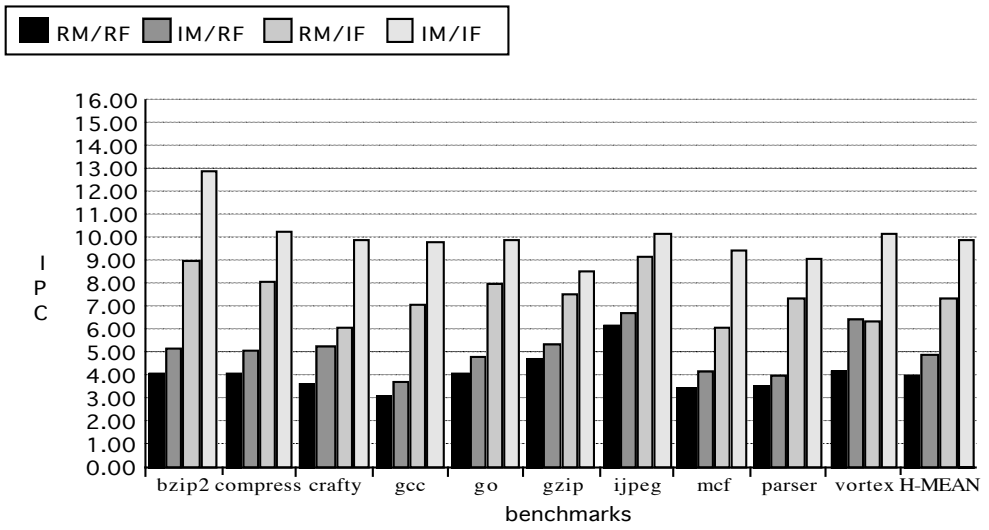

Figure 10.15: *IPC performance of the Levo machine geometry of 8-4-8 with relaxed memory latency and branch miss prediction.* The IPC performance of a Levo machine is shown for each of four cases of relaxed machine parameters across all benchmarks.

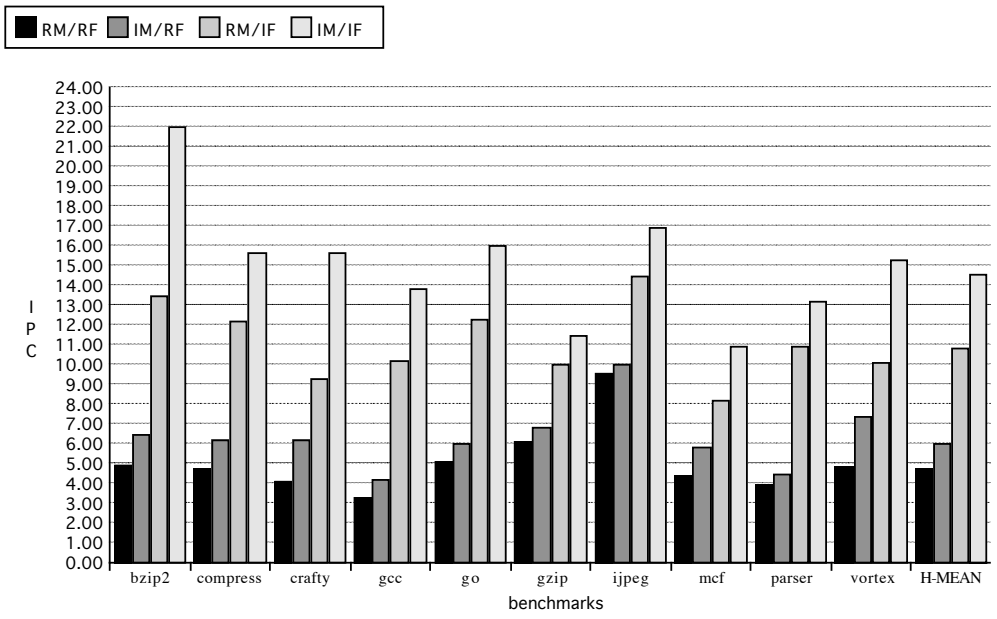

Figure 10.16: *IPC performance of the Levo machine geometry of 8-8-8 with relaxed memory latency and branch miss prediction.* The IPC performance of a Levo machine is shown for each of four cases of relaxed machine parameters across all benchmarks.

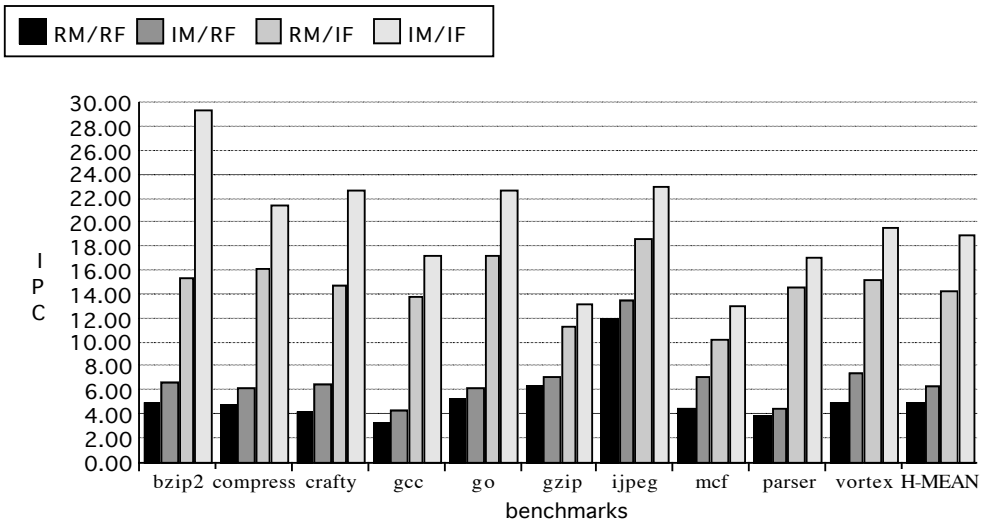

Figure 10.17: *IPC performance of the Levo machine geometry of 16-8-8 with relaxed memory latency and branch miss prediction.* The IPC performance of a Levo machine is shown for each of four cases of relaxed machine parameters across all benchmarks.

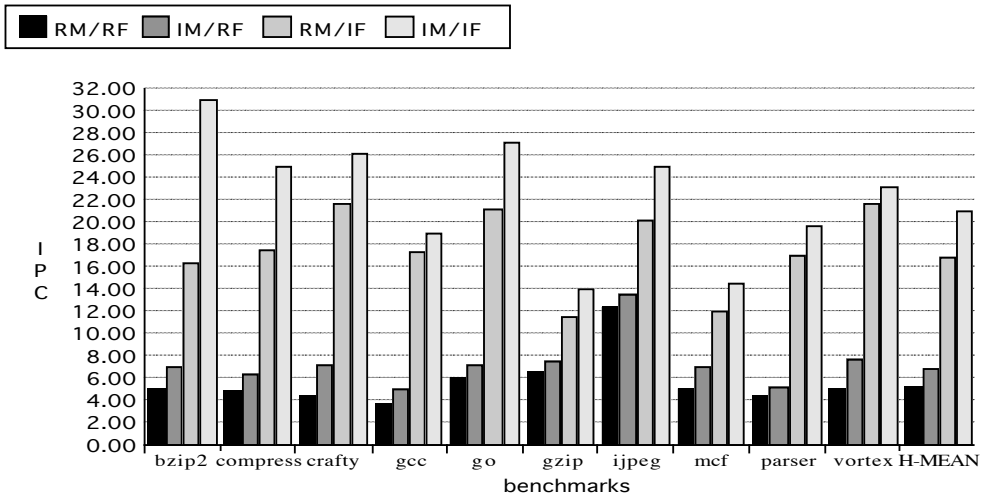

Figure 10.18: *IPC performance of the Levo machine geometry of 32-8-8 with relaxed memory latency and branch miss prediction.* The IPC performance of a Levo machine is shown for each of four cases of relaxed machine parameters across all benchmarks.

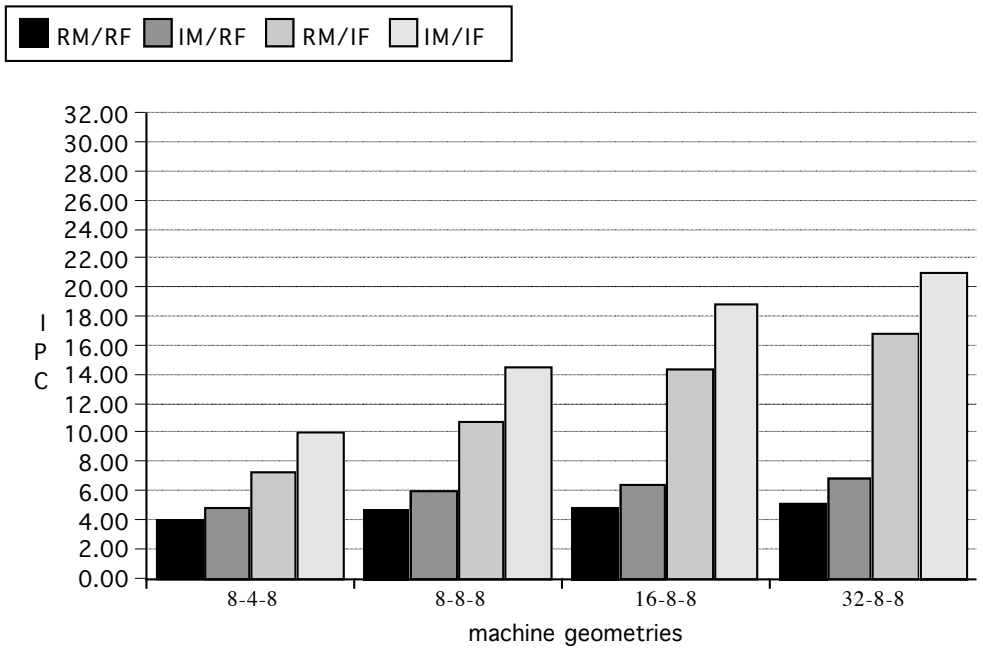

Figure 10.19: *IPC performance of four Levo machine geometries for each of four simulation cases.* The harmonic mean IPC performance (taken over all benchmark programs) for each of four Levo machine geometries is shown for each of four cases of relaxed machine parameters. The Levo machine geometries simulated are shown by their 3-tuples.

Chapter 11

Summary and Conclusions

We have introduced, developed, and evaluated several ideas related to a new microarchitectural approach for computing. Some of these ideas can be used and applied for novel processor designs alone, but all of them can also be used together synergistically. The goal of this research work was to explore a very novel approach for the microarchitectural organization and design of a processor that is suitable for exploiting the instruction level parallelism (ILP) within many existing program codes, for the purpose substantially increasing the execution performance of those program codes. Our focus was not on exploiting ILP in program codes that have a large degree of parallelism already, like program codes with explicitly parallel algorithms (often termed embarrassingly parallel). Rather our focus was design a machine that could extract ILP from those program codes that have historically been very difficult to parallelize in any way. Program codes of this sort are those that are single-threaded and exhibit a high degree of sequential dependencies within each part (and between parts) of the program. These program codes have often been termed integer sequential codes, and these are the very codes that we wanted to address with a new microarchitecture for the purpose of exploiting their inherent ILP to achieve performance gains.

Our goal also had the constraint that any new microarchitecture that we introduced had to be such that it could implement any existing instruction set architecture (ISA). The current general-purpose commercial computing market is not at all tolerant of either significant changes to existing architectures, or to any new architectures. The effort and resources required today to introduce a new machine architecture is generally now considered to be well beyond that of most commercial general-purpose computer companies. Too much has been invested in existing operating system ports and software to port them to new machine architectures. This has also been the case in the past, but is even more evident in today's computer marketplace. This constraint is most often expressed by the term that program codes need to be *binary compatible* with existing machines,

or that program codes must conform to an existing *application binary interface*. For this reason, we felt that any introduction of a new microarchitecture had to retain binary compatibility with any existing general-purpose machine architectures. We achieved this goal with our new microarchitecture, and further, our microarchitecture is likely binary-compatible with any new (yet unintroduced) machine architectures as well.

The constraint of program binary compatibility with existing machine architectures separates our work from most other research work that has explored the exploitation of ILP in the past. Most other research efforts that have focused on ILP extraction have also required either changes to the architecture or ISA of existing machines or the introduction of an entirely new machine architecture or ISA. The fact that most of these other research efforts have required either a substantially modified or completely new machine architecture has very dramatically limited the usefulness of these other research approaches. This has been the case because existing manufacturers of general-purpose computing machines do not want to take on the high cost (burden) of both introducing a new machine architecture along with all of the associated porting that would be required for the same, let alone the task of convincing software vendors and customers to port their software as well.

11.1 Summary and conclusions of the research work

In this section we summarize the novel ideas that we have introduced, defined, developed, or evaluated through simulation.

11.1.1 The Resource Flow execution model

By far the major and most novel part of this research work was the introduction of the Resource Flow execution model (presented in Chapter 3). We have defined this model of execution and have also defined and developed the foundational machine ideas and hardware components and subcomponents that facilitate the realization of this new approach towards program execution.

The basic philosophy behind the Resource Flow execution model is to only limit speculative machine execution by the structural hazards present in any given machine implementation (be it a large or a small machine in terms of its various component resources). This is a very different approach towards managing speculative execution as contrasted with that of conventional microarchitectures. Conventional machines limit the degree of speculative execution by several factors that are ignored or relaxed in the Resource Flow execution model. Specifically, conventional microarchitectural design approaches usually limit speculative execution to only occur after control dependencies and data dependencies among instructions has been already determined. Further,

speculative execution is also usually limited to only speculatively executing once before instruction retirement. The same instruction may execute (or speculatively execute again) but only usually after a complete flush of the machine execution window had been carried out. Additionally, conventional microarchitectural design approaches have to limit execution and speculative execution to the structural hazards present in the given machine (this being a constraint for which all microarchitectures obviously have to abide by). In contrast to conventional microarchitectural approaches, within the context of the Resource Flow approach only the structural hazards of the given machine need be adhered to. Besides that required constraint, all instructions within the execution window of a machine are allowed to compete with each other for execution and re-execution opportunities as they may arise through the pseudo random exchange of operands among the speculatively executing instructions. More specifically, in the Resource Flow execution approach, all instructions are permitted to attempt to execute (or re-execute) once they have either predicted or otherwise feasible operands available to them. No attempt is made to limit execution through the observance of an instruction's proper input data or control dependencies from preceding instructions in program-ordered time. Feasible operands are those that meet the architected requirement for being a possible operand (having the same architected memory address or register address) but which do not have to originate from the correct or proper preceding instruction in order to satisfy the present instruction operand inputs.

Although the Resource Flow execution model was presented in a way that embodied several microarchitectural components, the execution model presented did not in itself consist of a full machine microarchitecture (one which is complete enough to implement an entire actual processing machine). Nonetheless, much of what formed the basis of two complete microarchitectures that we subsequently introduced was already present in the basic description of the Resource Flow execution model.

Some of the ideas embodied within the Resource Flow idea includes the independent and localized execution of instructions (through the introduction of the issue station), the promotion of the idea of the operand to becoming a first class machine entity (at least as important as the idea of the instruction), and the development of how operands (of the various types) can be transfered among executing instructions in a way that allows for a low degree of coupling between independently executing instructions. This later point also being one that facilitates the expansion of a Resource Flow model microarchitecture in a physically scalable way (more on this later). Also inherent with the introduction and development of the Resource Flow execution model is that of dynamic data dependency determination at execution time rather than at instruction fetch or dispatch time. Some of these ideas are discussed further in the following subsections.

The primary microarchitectural component used to implement our idea of Resource Flow execution is that of the Issue Station (IS). This is a hardware machine component that extends the

idea of the reservation station or the issue window slot (as appropriate to the specific machine microarchitecture considered) in a way that allows the re-execution of an instruction dispatched to it without an intervening flush of the entire (or even partial) flush of the execution window. An instructions is dispatched to an issue station for the entire duration of the time that particular instruction (that dynamic instruction instance) is present within the execution window of the machine. Unlike either the reservation station or the issue window slot, the instruction does not vacate the issue station after an issue has been made to an execution unit or execution function unit. Rather, execution results are routed back to the originating issue station so that additional subsequent executions can occur as needed.

Several well defined interfaces (defined functional signal bus groups) to the issue station serve to both define its role in the machine and to isolate it from the other issue stations and from the rest of the machine as a whole. There are only three primary bus interfaces to each issue station. The first interface to the issue station provides the means to load it with a dispatched decoded instruction (along with its meta data such as instruction address and architected operand addresses). The second primary interface to the issue station is that of the operand forwarding and backwarding buses present in the execution window. The issue station both acquires its input operands and forwards its output operands using the operand forwarding bus group. The issue station requests that operands be forwarded to it using the operand backwarding bus group. And the third primary interface to the issue station are a set of two buses used to, respectively, issue operations to an execution unit and to retrieve the results back from that same execution unit once the specified execution is completed. The relative clarity and simplicity of each of these primary interfaces to the issue station is what contributes to and facilitates the possible physical scalability of microarchitectures based on the Resource Flow execution model.

A significant feature of the issue station is its relatively uniform handling of operands. As has been presented, operands are processed and managed within the issue station within operand blocks. These operand blocks serve to process both input and output operands for an instruction. The number of input and output operands blocks implemented within the issue station is dependent on the maximum number of each, respectively, that is possible with the given ISA.

11.1.2 The OpTiFlow microarchitecture

The OpTiFlow microarchitecture has been proposed, defined, and developed in the present work. This microarchitecture was designed to serve as an illustrative example for the implementation of a full microarchitecture that embodies the Resource Flow execution ideas. It was not intended to necessarily be the best or optimal microarchitecture based on the Resource Flow execution principles, but rather as one that could be used to validate the Resource Flow ideas as well as to

serve as a working example of a microarchitecture based on them. It has fulfilled these purposes.

Although the issue station is the primary component used in the implementation of the OpTiFlow microarchitecture, it also includes several other components that have been developed to implement it, making it a fully operational computer processing machine. Among these additional machine components are the OpTiFlow register file, the OpTiFlow load-store-queue (LSQ), and the OpTiFlow function unit (FU) components. The register file and load-store-queue are not like conventional counterparts since they require similar operand storage structures and operational management mechanisms that exist within the issue station component.

Once a representative microarchitecture (OpTiFlow) was designed and developed to show the working of the Resource Flow execution model, attention was turned towards its simulated evaluation. This was the item next addressed in this present research work.

11.1.3 The OpTiFlow simulator

As has been presented (in Chapter 5), a new simulation framework has been defined and developed for use in the simulation of the OpTiFlow microarchitecture (previously presented). The development of a new simulator framework was necessary due to the limitations of all existing research simulations frameworks at the outset of the present research. Unlike most existing simulation frameworks, this new simulation framework is well suited for the detailed simulation of the major machine components of the OpTiFlow microarchitecture at the register transfer level.

The software organization of this new simulator closely matched that of the hardware that it was supposed to simulate. The hardware structure of the OpTiFlow microarchitecture consists of hardware components that exhibit high cohesion of purpose within themselves, and at the same time a rather low coupling between components. In fact, it is this trait of the hardware structure and organization of the microarchitecture that was later exploited (in the present work) for making a physically scalable microarchitecture. This feature of the OpTiFlow microarchitecture was both noticed and exploited in the design and development of a simulator suitable for implementing it. These characteristics closely resemble the object oriented model of software programming and therefore an object oriented programming model was adopted and very naturally applied for implementation of each of the major hardware components making up the OpTiFlow microarchitecture.

This simulator was created and subsequently used to both validate the OpTiFlow microarchitecture (and the Resource Flow execution model in the process) as well as to perform an initial evaluation of the operation and performance of the OpTiFlow microarchitecture. Although any instruction set architecture could have been adopted for use for a representative OpTiFlow microarchitecture, the Alpha ISA was used due to the availability of an existing software definition of that ISA. The present research then turned to an actual initial evaluation of our OpTiFlow

microarchitecture using this newly developed simulator.

11.1.4 Evaluation of the OpTiFlow microarchitecture

Using the simulator specifically designed for the OpTiFlow microarchitecture (of Chapter 5), the validation of the OpTiFlow microarchitecture was accomplished. This work not only validated that the OpTiFlow microarchitecture properly implemented the Alpha ISA (as intended) but that it also correctly converged to program commitment while following the out-of-order execution and re-execution philosophy of the Resource Flow execution model. This validation also indirectly validated much of the whole Resource Flow execution model idea, since OpTiFlow is based on that execution model.

In additional to validating the correct operation of the OpTiFlow model to properly execute programs, three primary groups of experiments were carried out through the simulation of the OpTiFlow microarchitecture. The first group of experiments compared the OpTiFlow machine to a baseline conventional superscalar machine for a variety of machine configurations. For all comparisons, either identical or equivalent hardware machine resources were configured into each machine. Further, identical legacy binary programs were executed on each machine simulation model. The results showed that the OpTiFlow machine achieved an harmonic mean IPC speedup over the baseline conventional of approximately at least two. The second group of experiments explored the instruction re-execution behavior of the OpTiFlow machine and it revealed some weaknesses in the ability of the OpTiFlow machine to be performance scalable. The OpTiFlow machine is already not physically scalable (and therefore not performance scalable for this reason alone), but an additional scalability issue was observed. But due to the design nature and organization of the OpTiFlow microarchitecture (being a Resource Flow oriented microarchitecture to start with), both scalability issues can be overcome or mitigated with certain design enhancements. These design enhancements were are the subject of Chapters 8, 9, and 10 resulting in the definition, design, development, and evaluation of the Levo microarchitecture.

The third group of experiments explored the operational difference between two policies for managing instruction re-executions with the issue stations of the OpTiFlow machine (and thereby any Resource Flow based microarchitecture that uses anything like issue stations). The results showed that neither policy substantially performed better than the other, and neither one experienced substantially reduced instruction re-executions that might have led to lower power consumption.

11.1.5 Dynamic microarchitectural instruction predication

A new scheme for dynamic instruction predication implemented entirely within the microarchitecture of the machine has been designed and developed. Unlike the prior scheme (on which the

present scheme was derived), this new scheme allows for microarchitectural predication to be performed without the need for any centralized hardware component. This enhancement was required to achieve a fully distributed and physically scalable microarchitecture based on the Resource Flow execution model.

11.1.6 Operand filtering units

A means has been devised to allow for the physical scalability of a Resource Flow oriented microarchitecture (such as OpTiFlow) into larger physical sizes. Here a larger physical sized machine specifically refers to one which includes an increased number of individual hardware machine components. The primary machine component in view for the implementation of Resource Flow based microarchitectures (our intended goal) is the issue station. This work has resulted in the design and development of the various types of operand filter units, one for each type of operand: memory, register, and predicate.

The design of these operand filter units originated from the need to physically separate (electrically repeat) operand transfer buses, and it evolved from the original idea of using a simple pipelined register for that purpose. The requirement for physically repeating operand transfer buses did not originally envision the filtering of operands (which would not have been possible with a simple pipelined register), but evolved into a filtering role with our contribution to the original idea. The existing operand filter units not only repeat their operands from one physical bus to another but perform both a filtering function and a caching function for the type of operands that they handle. The contribution of filtering operands in addition to simply repeating them allowed for better utilization of operand transfer bus bandwidth and therefore contributes to an increased overall performance of a resulting physically scaled machine that uses them.

11.1.7 Characterization of register and memory access intervals

We performed a detailed characterization of the operand access intervals (as measured in instructions) for both architected register and memory operands. We evaluated three types of operand access intervals over both the register and memory operands. The three types of access intervals characterized were: a def-use interval, a useful-lifetime interval, and an access-use interval. Ten benchmark programs were used in the evaluation of all of these operand access intervals. The results for the register interval program characterizations validated the assumptions under which the we developed the register filter unit previously developed and introduced. We assumed that the length of register operand access intervals associated with the use of the register filtering unit were short enough so that operand transfer bus lengths between register filter units could be of reasonable size, defined appropriately by physical technological implementation considerations. Although

the characterization of the memory operand access intervals shows that they are much longer than those of register operands, our use of a memory operand filter unit (also developed in this present work) serves to partially address this characteristic of program execution. Further, since memory operands occur as a much reduced frequency than register operands do, any corresponding performance impact of having to traverse more than a single memory filter unit is not nearly as performance degrading as it would be in the register operand case.

11.1.8 The Levo microarchitecture

A physically distributed and scalable Resource Flow microarchitecture (named Levo) has been defined and developed. The physical scalability of this microarchitecture allows for the inclusion of many more hardware machine components that would otherwise be possible using conventional microarchitectures. This microarchitecture uses most of the previous machine mechanisms and components that have been developed, and which are all suited for implementing a scalable Resource Flow execution model. Some of these components and mechanisms include the issue station component idea, the various operand filter unit component ideas, and distributed dynamic microarchitectural predication. Further, the idea of localized resource sharing, allowing for distributed instruction execution, has been introduced with the sharing group (SG) mechanism.

We also evaluated the Levo microarchitecture through simulation. Simulation results show that the Levo machine achieves an harmonic mean IPC speedup of 2.03 over a conventional superscalar machine with similar hardware resources (a 103% performance improvement). Further simulations show that the Levo machine has much more performance potential when the instruction fetch branch prediction miss rate is relaxed to zero (no mispredictions). Harmonic mean IPC speedups of relaxed machine geometries over the corresponding baseline realistic machines range from 1.84 for a modest sized Levo machine (with geometry of `8-4-8`) to 3.26 for a large sized machine (with a geometry of `32-8-8`). These speedups are even large when L1 miss rate was relaxed also (but not nearly as dramatically). If progress can be made towards reducing instruction fetch branch mispredictions, very high IPCs (of over 20 for the `32-8-8` geometry) might be expected from larger sized Levo machines.

11.2 Summary of specific major research contributions

here we summarize the major specific contributions that we have made with the research that has been presented.

11.2.1 Development of the Resource Flow execution model

We have contributed to the overall definition and development of the Resource Flow execution model. This includes the basic idea of only constraining instruction execution to only the structural hazards of the given machine microarchitecture. Our contribution also includes the development of the way in which operands are requested by instructions and how they are forwarded.

11.2.2 Development of the issue station

One of our main contributions is in the definition and development of the issue station microarchitectural component. This component serves to implement much of the definition and requires of the Resource Flow execution model and also serves as the basic core building block for machine microarchitectures that are based on the Resource Flow execution model. Our contribution also includes the generalization of the operand block within the issue station.

11.2.3 Development of microarchitectural simulators

We defined and developed the microarchitectural simulator used for the evaluation of the OpTiFlow microarchitecture. We also contributed to the development of the microarchitectural simulator (named FastLevo) used for the evaluation of the Levo microarchitecture.

11.2.4 Validation and evaluation of the OpTiFlow microarchitecture

We both validated the proper working of the OpTiFlow microarchitecture as well as performed the first evaluation of that microarchitecture.

11.2.5 Design and development of the operand filter units

We designed and developed the various operand filter units that were introduced for the later development of a physically scalable microarchitecture (named Levo). Three different operand filter units were designed and developed, one each for: register operands, memory operands, and instruction predicate operands.

11.2.6 Characterization of register and memory operand access intervals

We performed the detailed characterization of various access intervals for both register and memory operands. The results of this characterization validated our design assumptions for the motivation and use of our operand filter units to facilitate a physically scalable microarchitecture.

11.2.7 Development of the Levo microarchitecture

We contributed to the development of the Levo microarchitecture. This is a physically scalable microarchitecture based on the principles of the Resource Flow execution model.

Bibliography

[1] Ahuja P.S., Skadron K., Martonosi M. and Clark D.W. Multipath execution: Opportunities and limits. In *Proceedings of the 12th International Conference on Supercomputing (ICS)*, New York, NY, July 1998. ACM Press.

[2] Allen J.R., Kennedy K., Portfield C., Warren J.D. Conversion of control dependence to data dependence. In *Proceedings of the 10th Annual Symposium on principles of programming languages*, pages 177–189, January 1983.

[3] Anderson D.W., Sparacio F. and Tomasulo F. The IBM/360 Model 91: Machine Philosophy and Instruction Handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

[4] Aragon J.L., Gonzalez J., Donzalez A. and Smith J.E. Dual path instruction processing. In *Proceedings of the 16th International Conference on Supercomputing (ICS)*, pages 220–229, New York, NY, 2002. ACM Press.

[5] Arvind, Nikhilpi R.S. Executing a origran on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 93(3):200–318, March 1990.

[6] Austin T.M. and Burger D. Simplescalar tutorial. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, November 1997.

[7] Bannon P., et al. Internal architecture of Alpha 21164 microprocessor. In *Proceedings of COMPCON*, pages 79–87, March 1995.

[8] Doug Burger, James R. Goodman, and Alain Kägi. Limited bandwidth to affect processor design. *IEEE Micro*, 17(6):55–62, November 1997.

[9] Burger D., Austin T.M. The SimpleScalar toolset, version 2.0. Technical Report UWMADISONCS CS-TR-95-1342, University of Wiconsin-Madison, Madison, WI, June 1995.

[10] Burger D., James R., Goodman J.R., and Kagi A. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, pages 78–89, New York, NY, 1996. ACM, ACM Press.

[11] Burger D.C., Goodman J.R., and Kagi A. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report UWMADISONCS CS-TR-95-1261, University of Wisconsin-Madison, Madison, WI, January 1995.

[12] Calder B., Reinman G. and Tullsen D.M. Selective value prediction. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, pages 64–74, New York, NY, 1999. ACM Press.

[13] Chang P-Y., Hao E., Patt Y. Target prediction for indirect jumps. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 274–283, Denver, CO, Jun 1997. ACM Press.

[14] Chen I.K., Coffey J.T. and Mudge T. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 128–137, Cambridge, MA, October 1996.

[15] Cher C. and Vijaykumar T.N. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, New York, NY, Nov 2001. ACM Press.

[16] Chuang W., Valder B. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 183–192, June 2003.

[17] Cleary J.G, Pearson M.W. and Kinawi H. The architecture of an optimistic CPU: The Warp Engine. In *Proceedings of the Hawaii Internationl Conference on System Science*, pages 163–172, January 1995.

[18] Collins J.D., Tullen D.M. and Wang H. Control flow optimization via dynamic recovergence prediction. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, Dec 2004.

[19] Conners W.D., Florkiowski J. and Patton S.K. The IBM 3033: An inside look. *Datamation*, pages 198–218, May 1979.

[20] Control Data Corporation, Palo Alto, CA. *6400/6500/6600 Computer Systems: COMPAS Reference Manual*, 1969.

[21] Davidson J., and Jinturkar S. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO)*, pages 125–132, New York, NY, 1995. ACM Press.

[22] Dehnert J.C., Hsu P.Y., Bratt J.P. Overlapped loop support in the Cydra 5. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–38, New York, NY, April 1989. ACM Press.

[23] Delattre F. and Prieur M. Intel Core 2 Duo - Test, July 2006.

[24] Dennis J. and Misunas D. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd International Symposium on Computer Architecture (ISCA)*, pages 126–132, January 1975.

[25] Driesen K. and Hlzle U. Accurate indirect branch prediction. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, Barcelona, Spain, July 1998. ACM Press.

[26] Driesen K. and Holzle U. The cascaded predictor: Economic and adaptive branch target prediction. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO)*, Dallas, TX, November 1998. ACM Press.

[27] Eeckhout L. and Bosschere K.D. Hybrid analytical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques PACT*, New York, NY, September 2001. IEEE Press.

[28] Eustace A. and Srivastava A. ATOM: A flexible interface for building high performance program analysis tools. Technical Report WRL-TN-44, Digital Western Research Lab, Palo Alto, CA, July 1994.

[29] Eustace A. and Srivastava A. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.

[30] Farkas K.I., Chow P., Jouppi N.P., Vranesic Z. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pages 149–159, 1997.

[31] Ferrante J., Ottenstein K., and Warren J. The Program Dependence Graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[32] Franklin M. The multiscalar architecture. Technical Report CS-TR-1993-1196, University of Wisconsin-Madison, Madison, WI, 1993.

[33] Franklin M. and Sohi G.S. Register traffic analysis for streamlining inter-operation communication in fine-grained parallel processors. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO)*, pages 236–245, New York, NY, Dec 1992. ACM Press.

[34] Franklin M. and Sohi G.S. ARB: A hardware mechanism for dynamic memory disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[35] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedingsof the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[36] Gloy N., Young C., Chen J.B. and Smith M.D. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, pages 12–21, New York, NY, 1996. ACM Press.

[37] Gonzalez A. and Marcuello P. Dependence speculative multithreaded architecture. Technical report, Universitat Polytecnica de Catalunya, 1998.

[38] Gonzalez J. Evaluation of alternative data speculation approaches for superscalar processors, 1997.

[39] Gonzalez J. and Gonzalez A. Limits on instruction-level parallelism with data speculation. Technical Report UPC-DAC-1997-34, UPC, Barcelona, Spain, 1997.

[40] Gonzalez J., and Gonzalez A. Data value speculation in superscalar processors, 1998.

[41] Gonzalez J. and Gonzalez A. Limits of Instruction Level Parallelism with Data Speculation. In *Proceedings of the VECPAR Conference*, pages 585–598, 1998.

[42] Gonzalez J. and Gonzalez A. The potential of data value speculation to boost ILP. In *Proceedings of the 12th International Conference on Supercomputing (ICS)*, pages 21–27. ACM, 1998.

[43] Gopal S., Vijaykumar T.N., Smith J.E., Sohi G.S. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*. IEEE, Feb 1998.

[44] Heil T.H. and Smith J.E. Selective dual path execution. Technical report, University of Wisconsin-Madison, Madison, WI, 1996.

[45] Hennessy J.L. and Patterson D.A. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann, Palo Alto, CA, 1995.

[46] Henry D.S and Kuszmaul B.C. and Loh G.H. and Sami R. Circuits for Wide-Window Superscalar Processors. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 236–247, New York, NY, Jun 2000. ACM, ACM Press.

[47] Henry D.S., Kuszmaul B.C., Viswanath V. The ultrascalar processor – an asymptotically scalable superscalar microarchitecture. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*. IEEE, Mar 1999.

[48] Herrod S.A. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.

[49] Hinton G., Sager D., Upton M., Boggs D., Carmean D., Kyker A. and Roussel P. The microarchitecture of the Pentium-4 processor. *Intel Technical Journal - Q1*, 2001.

[50] Hodgin R. What is a processor's pipeline?, February 2006.

[51] Hughes R. Intel Pentium 4 Prescott processor, 2006.

[52] Ibrahim K.Z., Byrd G.T. and Rotenberg E. Slipstream execution mode for CMP-Based multiprocessors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*. IEEE, Feb 2003.

[53] Intel. *Intel Itanium-2 Processor Reference Manual for Software Development and Optimization*, June 2002.

[54] Intel Corp. *iA-64 Application Developer's Architecture Guide*, 1999.

[55] Intel Corporation. Intel 64 and ia-32 architectures software develop's manual. 253665-022US, November 2006.

[56] Jacobsen E., Rotenberg E. and Smith J.E. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO)*, pages 142–152, New York, NY, 1996. ACM Press.

[57] Jain I.A. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading, 2001.

[58] Jiser J., Carr S. and Sweany P. Global register partitioning. In *Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New York, NY, October 2000. IEEE Press.

[59] Jouppi N.P. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, 1989.

[60] Kaeli D. and Emma P. Improving the accuracy of history-based branch prediction. *IEEE Transactions on Computers*, 46(4):469–472, April 1997.

[61] Kaeli D., Morano D.A. and Uht A. Preserving dependencies in a large-scale distributed microarchitecture. Technical Report TR 022002-001, University of Rhode Island, Dec 2001.

[62] Kaeli D.R. and Emma P.G. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19-3, pages 34–42, New York, NY, 1991. ACM Press.

[63] Kahlafi A. private communication, November 2000.

[64] John Kalamatianos. *Microarchitectural and Compile-Time Optimizations for Performance Improvement of Procedural and Object-Oriented Languages.* PhD thesis, Northeastern University, 2000.

[65] Kalla R., Sinharoy B., and Tendler J.M. The power 5 chip: a dual core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[66] Kemp G.A. and Franklin M. PEWs: A decentralized dynamic scheduler for ILP processing. In *Proceedings of the 24th International Conference on Parallel Computing*, pages 239–246, 1996.

[67] Kessler R.E. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[68] Kessler R.E., McLellan E.J., and Webb D.A. The Alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design (ICCD)*, October 1998.

[69] Khalafi A. Exploring multipath execution on a distributed microarchitece. Northeastern University, May 2003.

[70] Kim H., Mutlo O., Stark J., and Patt Y.N. Wish branches: combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*, pages 43–45, New York, NY, November 2005. ACM Press.

[71] Klauser A.S. and Grunwald D. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO)*, New York, NY, Nov 1999. ACM Press.

[72] Klauser A.S., Austin T., Grunwald D., and Calder B. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New York, NY, October 1998. ACM Press.

[73] Klauser A.S., Austin T., Grunwald D., Calder B. Restricted multipath execution using dynamic predication. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 278–285, New York, NY, October 1998. ACM Press.

[74] Klauser A.S., Pathankar A. and Grunwald D. Selective eager execution on the polypath architecture. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, pages 250–259, New York, NY, Jun 1998. ACM Press.

[75] Kobayashi M. Dynamic characteristics of loops. *IEEE Transactions on Computers*, 33(2):125–132, 1984.

[76] Krewell K. Intel's McKinley Comes Into View. *Cahners Microprocessor*, 15(10):1–5, Oct 2001.

[77] Lam M. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 23, pages 318–328, 1988.

[78] Lam M.S. and Wilson R.P. Limits of control flow on parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pages 46–57, New York, NY, May 1992. ACM, ACM Press.

[79] Lee J. and Smith A. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1), January 1984.

[80] Leibholz D. and Razdan R. The Alpha 21264: A 500 MHz out-of-order execution microprocessor. In *Proceedings of COMPCON*, pages 28–36, 1997.

[81] Lipasti M.H. Value locality and speculative execution. Ph.D. thesis, Carnegie Mellon University, April 1997.

[82] Lipasti M.H. and Shen J.P. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO)*, pages 226–237, 1996.

[83] Lipasti M.H. and Shen J.P. Approaching 10 IPC via superspeculation. Technical Report CMU-MIG-1, Carnegie Mellon University, 1997.

[84] Lipasti M.H. and Shen J.P. The performance potential of value and dependence prediction. In *European Conference on Parallel Processing*, pages 1043–1052, 1997.

[85] Lipasti M.H. and Shen J.P. Superspeculative microarchitecture for beyond AD 2000. *IEEE Computer*, 30(9):59–66, 1997.

[86] Lipasti M.H. and Shen J.P. Exploiting value locality to exceed the dataflow limit. *International Journal of Parallel Programming*, 26(4):505–538, 1998.

[87] Lipasti M.H., Wilkerson C.B. and Shen J.P. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 138–147, 1996.

[88] Lowney P.G., Freudenberger S.M., Karzes T.J., Lichtenstein W.D., Nix R.P., O'Donnell J.S., Ruttenberg J.C. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1):51–142, May 1993.

[89] Ludloff C. IA-32 Implementation, Intel P4. http://www.sandpile.org/impl/p4.htm, Jan 2002.

[90] Madison A., Bunt R. Characteristics of program localities. *Communications of the ACM*, May 1976.

[91] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO)*, pages 217–227, November 1994.

[92] Mahlke S.A., Chen W.Y., Gyllenhaal J.C., Hwu W., Chang P., and Kiyohara T. Compiler code transformations for superscalar-based high-performance systems. In *Proceedings of Supercomputing*, pages 808–817, Minneapolis, MN, 1992. IEEE Press.

[93] McFarling S. Combining branch predictors. Technical Report TN-36, DEC Western Research Laboratory, Jun 1993.

[94] McKinley K., Carr S., and Tseng C-W. Improving data locality with loop transformations. *ACM Transactions in Programming Languages and Systems*, 18(4):424–453, 1996.

[95] Morano D.A. Execution-time Instruction Predication. Technical Report TR 032002-0100, University of Rhode Island, Mar 2002.

[96] Morano D.A., Khalafi A., Kaeli D.R., Uht A.K. Realizing high IPC through a scalable memory-latency tolerant multipath microarchitecture. In *Proceedings of the MEDEA Workshop (held in conjuction with PACT'02)*, 2002.

[97] Nagarajan R., Sankaralingam K., Burger D. and Keckler S.W. A design space evaluation of grid processor architectures. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, New York, NY, Nov 2001. ACM Press.

[98] Palacharla S., Jouppi N.P. and Smith J.E. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 206–218, New York, NY, 1997. ACM Press.

[99] Pan S.-T., So K. and Rahmeh J.T. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 76–84, New York, NY, 1992. ACM Press.

[100] Papworth D.B. Tuning the Pentium Pro microarchitecture. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO)*, volume 16, pages 8–15. IEEE, IEEE Press, Apr 1996.

[101] Popescu V., Schultz M., Spracklen J., Gibson G., Lightner B. and Isaman D. The Metaflow architecture. *IEEE Micro*, 11(3):10–13, 63–73, 1991.

[102] Preston R.P., Badeau R.W., Bailey D.W., Bell S.L., Biro L.L., Bowhill W.J., Dever D.E., Felix S., Gammack R., Germini V., Gowan M.K., Gronowski P., Jackson D.B., Mehta S., Morton S.V., Pickholtz J.D., Reilly N.H., Smith M.J. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *Proceedings of the International Solid State Circuits Conference*, Jan 2002.

[103] Raasch S.E., Binkert N.L. Reinhardt S.K. A Scalable Intruction Queue Using Dependence Chains. In *Proceedings of 29th International Symposium on Computer Architecture (ISCA)*. ACM, May 2002.

[104] Ranganathan N. and Franklin M. An empirical study of decentralized ILP execution models. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 272–281, New York, NY, October 1998. ACM Press.

[105] Rau B.R. Dynamic scheduling techniqueues for CLIW processors. Technical Report HPL-93-52, HP Labs, June 1993.

[106] Rau B.R., Yen D.W., Yem W., Towie R.A. The Cydra 5 depeartmental supercomputer: design philospohies, decisions, and trade-offs. *Computer*, 22(1):12–26, 28–30, 32–35, January 1989.

[107] Riseman E.M. and Foster C.C. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, 21(12):1405–1411, December 1972.

[108] Rosenblum M., Bugnion E., Devine S. and Herrod S. Using the SimOS machine simulator to study complex computer systems. ACM TOMACS Special Issue on Computer Simulation, 1997.

[109] Rosenblum M., Herrod S.A., Witchel E. and Gupta A. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 1995.

[110] Rotenberg E. and Smith J.E. Control independence in trace processors. In *Proceedings of the 32th International Symposium on Microarchitecture (MICRO)*, 1999.

[111] Rotenberg E., Jacobsom Q. and Sazeides Y. Trace processors. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pages 138–148, 1997.

[112] Russell R. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[113] Rychlik B., Faistl J., Krug B., Kurland A., Jung J., Velev M. and Shen J. Efficient and accurate value prediction using dynamic classification. Technical report, Carnegie Mellon University, 1998.

[114] Sair S., and Kaeli D. A study of loop unrolling for vliw-based dsp processor. Proceedings of the Workshop on Signal Processing Systems, October 1998.

[115] Sankaranarayanan K. and Skadron K. A Scheme for Selective Squash and Re-issue for Single-Sided Branch Hammocks. Technical Report CS-2001-14, University of Virginia, Charlottesville, VA, July 2001.

[116] Sankaranarayanan K. and Skadron K. A scheme for selective squash and re-issue for single-sided branch Hammocks. IEEE Press, October 2001.

[117] Sazeides Y. And Smith J.E. Implementations of context based value predictors. Technical Report TRECE 97-8, University of Wisconsin-Madison, Madison, WI, December 1997.

[118] Sazeides Y. and Smith J.E. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pages 248–258, New York, NY, 1997. ACM Press.

[119] Schlansker M.S. and Rau B.R. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, Feb 2000.

[120] Seznec A., Felix S., Krishnan V., Sazeides Y. Design tradeoffs for the alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, pages 25–29, May 2002.

[121] Sherwood T. and Calder B. Loop termination prediction. In *Proceedings of the 3rd International Symposim on High Performance Computing*. Springer-Verlag, October 2000.

[122] Sherwood T., Hamerly G. and Calder B. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, New York, NY, May 2002. ACM Press.

[123] Sherwood T., Perelman E., Hamerly G., Calder B. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 138–147, October 2002.

[124] Sites R.L. and Witek R.T. *Alpha AXP Architecture Refernce Manual*. Digital Equipment Corporation, 1995.

[125] Skadron K. and Ahuja P.S. HydraScalar: A Multipath-Capable Simulator. *IEEE Technical Committee on Computer Architecture Newsletter*, 2001.

[126] Smelyanskiy M., Tyson G.S. and Davidson E.S. Register queues: A new hardware/software approach to efficient software pipelining. In *Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New York, NY, October 2000. IEEE Press.

[127] Smith J.E. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium in Computer Architecture (ISCA)*, pages 135–148, Minneapolis, MN, 1981.

[128] Smith J.E. and Pleszkun A.R. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture (ISCA)*, pages 36–44, Los Alamitos, CA, 1985. IEEE Computer Society Press.

[129] Smith J.E. and Pleszkun A.R. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, September 1988.

[130] Smith J.E. and Sohi G.S. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83:1609–1624, Dec 1995.

[131] Sohi G.S. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. In *IEEE Transactions of Computers*, volume 39, pages 349–359. IEEE Press, June 1990.

[132] Sohi G.S. and Vajapeyam G. Instruction issue logic for high-performance interruptable pipelined processors. In *Proceedings of the 14th International Symposium on Computer Architecture (ISCA)*, pages 27–34, New York, NY, June 1987. ACM Press.

[133] Sohi G.S., Breach S. and Vijaykumar T.N. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*, pages 414–425, New York, NY, Jun 1995. ACM Press.

[134] Srivastava A. and Eustace A. ATOM: A system for building customized program analysis tools. Technical Report WRL-TR-94.2, Digital Western Research Lab, Palo Alto, CA, Mar 1994.

[135] Stokes J. Inside AMD's Hammer: the 64-bit architecture behind the Opteron and Athlon 64, February 2005.

[136] Sundararaman K.K. and Franklin M. Multiscalar execution along a single flow of control. In *Proceedings of the International Conference on Parallel Computing*, pages 106–113, 1997.

[137] Taylor M.B., Kim J., Miller J., Wentzlaff D., Ghodrat F., Greenwald B., Hoffmann H., Johnson J., Lee J., Lee W., Ma A., Saraf A., Seneski M., Shnidman N., Strumpen V., Frank M., Amarasinghe S. and Agarwal A. The RAW microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 2002.

[138] Thomas R. and Franklin M. Using dataflow based context for accurate value prediction. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[139] Thornton J.E. Parallel operation in the Control Data 6600. In *Proceedings of the AFIPS*, volume 26, pages 33–40, 1964.

[140] Tjaden G.S. and Flynn M.J. Representation of concurrency with ordering matrices. In *Proceedings of COMPCON*, volume C-22, pages 752–761, Aug 1973.

[141] Tomasulo R.M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[142] Tsai J-Y. and Yew P-C. The Superthreaded Architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 35–46, 1996.

[143] Tubella J. and Gonzalez A. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture (HPCA)*, Las Vegas, NV, January 1998.

[144] Tullsem D.M., Eggers S.J., Levy H.M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*. ACM, June 1995.

[145] Tullsen D.M., Eggers S.J., Emer J.S., Levy H.M., Lo J.L. and Stamm R.L. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23th International Symposium on Computer Architecture (ISCA*, pages 191–202, New York, NY, May 1996. ACM Press.

[146] Tyson G., Lick K. and Farrens M. Limited duel path execution. Technical Report CSE-TR 346-97, University of Michigan, 1997.

[147] Gang-Ryung Uh and David B. Whalley. Effectively exploiting indirect jumps. *Software - Practice and Experience*, 29(12):1061–1101, 1999.

[148] Uht A. K. and Sindagi V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proceedings of the 20th International Symposium on Microarchitecture (MICRO)*, pages 313–325. ACM, November 1995.

[149] Uht A., Morano D., Khalafi A., Alba M. and Kaeli D. Realizing high IPC using time-tagged resource-flow computing. In *EUROPAR Conference*, Paderborn, Germany, August 2002.

[150] Uht A.K. An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, pages 41–50. University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, January 1986. Second Place for Best Paper in Computer Architecture Track.

[151] Uht A.K. Extraction of massive instruction level parallelism. Technical Report 1292-0001, Dept. of EE, URI, Dec 1992.

[152] Uht A.K. Verification of ILP speedups in the 10s for Disjoint Eager Execution. Technical Report TR 0697-0001 Rev. A, University of Rhode Island, July 1997.

[153] Uht A.K., Morano D., Khalafi A., and Kaeli D. Levo – a scalable processor with high IPC. *Journal of Instruction Level Parallelism*, 5, August 2003.

[154] Uht A.K., Morano D.A., Khalafi A., de Alba M., Kaeli D. Realizing high IPC using time-tagged resource-flow computing. In *Proceedings of the the EUROPAR Conference*, Aug 2002.

[155] Uht A.K., Morano D.A., Khalafi A., de Alba M., Wenisch T., Ashouei M. and Kaeli D. IPC in the 10's via resource flow computing with levo. Technical Report TR 092001-001, University of Rhode Island, September 2001.

[156] Vajapeyam S., Mitra T. Improving superscalar intruction dispatch and issue by exploiting dyanmic code sequences. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, New York, NY, 1997. ACM Press.

[157] Veenstra J.E. and Fowler F.J. MINT tutorial and user manual. Technical Report 452, Computer Science Department, The University of Rochester, June 1993.

[158] Veenstra J.E. and Fowler R.J. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207, January 1994.

[159] Verkamo A. Emperical results on locality in database referencing. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Aug 1985.

[160] Vidyadhar P. and Bhaskarpillia G. An inter-reference gap model for temporal locaility in program behavior. In *Proceedings of SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 191–300, New York, NY, 1995. ACM Press.

[161] Waingold E., Taylor M., Srikrishna D., Sarkar V., Lee W., Lee V., Kim J., Frank M., Finch P., Barua R., Babb J., Amarasinghe S. and Agarwal A. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, 1997.

[162] Wall D.W. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, April 1991. ACM Press.

[163] Wall D.W. Limits of instruction level parallelism. Technical Report TR 96/6, DEC Western Research Laboratory, November 1993.

[164] Wallace S., Calder B. and Tullsen D.M. Threaded multiple path execution. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, New York, NY, Jun 1998. ACM Press.

[165] Wang P.H., Wang H., Kling R.M., Ramakrishnan K., and Sten J.P. Register renaming and scheduling dor dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–25, January 2001.

[166] Wang S.S.H. and Uht A.K. Ideograph/ideogram: Framework/architecture for eager execution. In *Proceedings of the 23rd Symposium and Workshop on Microprogramming and Microarchitecture*, pages 125–134, New York, NY, Nov 1990. ACM Press.

[167] Warter N.J., Lavery D.M., Hwu W.W. The benefit of predicated execution for software pipelining. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 497–506, January 1993.

[168] Witchel E. and Rosenblum M. Embra: Fast and flexible machine simulation. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA., 1996.

[169] Yeager K.C. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, Apr 1996.

[170] Yeh T.-Y. and Patt Y.N. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 257–266, Goteborg, Sweden, 1993.

[171] Yeh T.Y. and Patt Y.N. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pages 124–134, Gold Coast, Australia, 1992. ACM and IEEE Computer Society.