Contents

0-8493-0052-5/00/$0.00+$.50

List of Tables

0-8493-0052-5/00/$0.00+$.50

List of Figures

0-8493-0052-5/00/$0.00+$.50

Part I

This is a Part

Chapter 15

Resource Flow Microarchitectures

David A. Morano
Northeastern University

15.1 Introduction

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 complex. This complexity has limited or discouraged the use of more advanced speculation techniques such as value prediction, multipath execution, and the

0-8493-0052-5/00/$0.00+$.50

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 correct results gets undesirably larger also.

A microarchitectural approach oriented towards handling speculative execution in a more general and uniform way is presented. The 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, 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. The microarchitectural approach presented also lends itself towards resource scalability through manageable spatial distribution of machine components. This last objective is also realized through the rather general and uniform way in which instructions and operands are handled.

This 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. Further, the importance of instruction operands (whether they be control or data) is elevated to almost the level of importance of an instruction itself. Operands are enhanced with additional state that allows for a more uniform 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 basic concepts employed in existing Resource Flow microarchitectures are presented along with the various structures needed to manage this aggressive form of speculative execution in a generalized way. Also described is how a variety of aggressive speculative schemes and ideas fit into this new microarchitectural approach more easily than if they were considered and implemented without this generalized instruction and operand management. Finally, two representative example microarchitectures, based on the Resource Flow concepts, are briefly presented. The first of these is oriented for a silicon area size somewhere between previous or existing processors such as the Alpha EV8 [15], the larger Pentium-4 [6] variants, or the Power-4 (in the 100 to 200 million transistor range) and next generation processors (maybe 200 to 300 million transistors). The second microarchitecture presented illustrates where the Resource Flow ideas could lead with much larger machine sizes

and component resources (approaching 1 billion transistors and beyond). As will be seen, the physical scalability of this microarchitecture is substantially facilitated by the basic Resource Flow design concepts.

15.2 Motivation for More Parallelism

Although many high performance applications today can be parallelized at the source code level and executed on symmetric multiprocessors or clustered systems, there are and will continue to be requirements for achieving the highest performance on single threaded program codes. An attempt is made to target this application problem space through the extraction of instruction level parallelism (ILP). However, high execution performance through ILP extraction has not generally been achieved even though large amounts of ILP are present in integer sequential programs codes. 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 (e.g., SpecInt-2000). The work of researchers including Lam and Wilson [10], Uht and Sindagi [26], Gonzalez and Gonzalez [5] have shown that there exists a great amount of instruction level parallelism that is not being exploited by any existing computer designs.

A basic challenge is to find program parallelism and then allow execution to occur speculatively, in parallel, and out of order over a large number of instructions. Generally, this is achieved by introducing multiple execution units into the microarchitecture where each unit can operate as independently as possible and in parallel, thus achieving increased executed instructions per clock (IPC). Since it is also usually very desirable (if not critical) to support legacy instruction set architectures (ISAs), this should also be a goal when pursuing enhanced high IPC mechanisms. For this reason, a microarchitecture that is suitable for implementing any ISA is explored.

Microarchitectures that have employed the use of multiple execution units are the Multiscalar-like processors [20, 21], the SuperThreaded processor model [24], and the Parallel Execution Window processor model [9]. Other microarchitecture proposals such as the MultiCluster machine model by Farkas et al. [2] 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. [11] They achieved an IPC of about 7 with conventional hardware assumptions but also by using value prediction. Nagarajan proposed a *Grid Architecture* of ALUs connected by an operand network. [14] This has some similarities to the present work. However, unlike

the present work, their microarchitecture relies on the coordinated use of the compiler along with a new ISA to obtain higher IPCs.

The present goal is to combine many of the design features of proposed speculative microarchitectures into a more generalized form. These include: control-flow prediction, value prediction, dynamic microarchitectural instruction predication, dynamic operand dependency determination, and multipath execution. Another objective is to try to facilitate a substantially increased sized processor along with many more resources for large-scale simultaneous speculative instruction execution, and therefore greater possible ILP extraction. This last objective is illustrated with a discussion of the second representative microarchitecture below, which shows how large physical processor scalability might possibly be achieved.

15.3 Resource Flow Basic Concepts

In the following sections several of the basic concepts embodied in the Resource Flow execution model are presented. In this 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. Also, the establishment of a machine component used to hold an instruction during its entire lifecycle after it has been dispatched for possible execution is presented in some detail. This component will generalize how instructions are handled during both initial execution and subsequent executions as well as eliminate the need for any additional re-order components. This component is termed an *active station* (AS). Additionally, some details about how operands are passed from one instruction to another is discussed.

15.3.1 The Operand as a First Class Entity

Within the presented microarchitectural framework, three types of instruction operands that need special handling can be identified. These are termed *register*, *memory*, and *predicate*. Although register and memory operands are both data operands, they are different enough is their use and characteristics that they need to be treated somewhat differently at the microarchitectural level of the machine. For example, register operands (generally) have fixed addresses associated with them (their address is not usually dynamically determined by the instruction during its execution) while it is very common for memory operands to have their addresses determined by a a memory-accessing instruction during the execution of the instruction. Often the address of a

memory operand is calculated at execution time through some manipulation of a constant and one or more register operands. Older ISAs (such as the Digital Equipment VAX) could even determine a memory address from a memory location. Further, the number of registers within a given ISA tends to be very small (like 32 to 256 or so) while the number of memory locations is rapidly being standardized on two to the power 64. These differences therefore warrant different hardware treatment within the machine.

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 programmer. A predicate operand of this type is present in an ISA such as the iA-64, for example. [7] For the present purpose, 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 handling of microarchitectural predicate operands that are not a part of the ISA and are therefore not visible to the programmer. This latter type of predication is entirely maintained by the microarchitecture itself and 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, at any time, any instruction can always still be allowed to execute. The only issue is whether the output result can become part of the program committed state. In the Resource Flow model, microarchitecture-only predicate operands share some similarities to register and memory (data) operands but are still different enough that they warrant some additional special treatment. Dynamic microarchitectural instruction predication is employed, whether or not the ISA defines explicit predicate registers.

15.3.2 Dynamic Dependency Ordering

Rather then calculating instruction dependencies at instruction dispatch or issue time, instructions are allowed 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 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 an instruction or an operand with respect to each other 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 [1] also used time-tags to manage large amounts of speculative execution, but the present use of time-tags is substantially simpler than theirs.

A time-tag value of zero (in the simplest case) 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. As instructions retire, the time-tag registers 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. Negative valued tags can also have microarchitectural applications such as when when memory operands are committed but still need to be snooped in a store queue. Output operands, created as a result of an instruction's execution, take on the same time-tag value as its originating instruction. By comparing time-tag values with each other, the relative program-ordered time relationship is determined. This use of time-tags for dependency ordering obviates the need for more complicated microarchitectural structures such as dependency matrices. [22]

15.3.3 Handling Multipath Execution

Multipath execution is a means to speculatively execute down more than one future path of a program simultaneously (see Chapter 6). In order to provide proper dependency ordering for multipath execution an additional register tag (termed a *path ID*) is introduced that will be associated with both instructions and operands. Generally, a new speculative path may be formed after each conditional branch is encountered within in the instructions 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). One option is to dynamically follow each output path and assign successively higher time-tag values to succeeding instructions on each path independently of the other. The second option is to try to determine if the *taken* output path of the branch joins with the *not-taken* output path instruction stream. If a join is determined, the instruction following the *not-taken* output path is assigned a time value one higher than the branch itself, while the first instruction on the *taken* path is assigned the same value as the instruction at the join as it exists on the not-taken path of the branch.

Note that both options may be employed simultaneously in a microarchitecture. Details on when each of these choices might be made depends on whether it is desirable to specially handle speculative path joins in order to take further advantage of control-independent instructions beyond the join point. This and many other details concerning multipath execution is an open area of ongoing research. In either case, there may exist instructions in flight on separate speculative paths 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.

15.3.4 Names and Renaming

For instructions, names for them 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

Generally, the type of the operand would be *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 predicate operands, the address may be absent entirely for some implementations, or might be some address that is used within the microarchitecture to further identify the particular predicate register in question. Microarchitecture predication schemes that have used both approaches have been devised.

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 provided for. This information can be used to eliminate the need for physical register renaming, register update units, or reorder buffers.

15.3.5 The Active Station Idea

The introduction of the active station component and its operation is what most distinguishes the present execution model from that of most others. The active station and its operational philosophy has been previously introduced by Uht et al. [28] These active stations can be thought of as being reservation stations [23] but with additional state and logic added 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 issued to a function unit (FU). Also, unlike reservation stations, but like an issue window slot, the instruction operand from an active station may be issued to different function units (not just the one that is strictly associated with a reservation station). With a conventional reservation station, the outputs of the execution are looped back to both waiting reservation stations and to the architected register file. That does not occur with action stations. Rather, output results from executions return to the originating active station and they are then further distributed from there with additional identifying information added to them. Having execution units forward results directly is possible but has not been explored due to the added complexity of the bus interconnections needed. In any event, the active must still acquire output results in the event that those results need to be broadcast to other ASes subsequently due to control or data flow dependency changes.

The state associated with an active 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 instruction specific state is used to guide the instruction through its life cycle until retirement, while the remaining state consists of zero or more input source operands and one or more output destination operands. All operands regardless of type and whether source or destination occupy a similar structure within an active station, 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* is used as part of the dynamic predication scheme used. [12] 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 additional *execution state* of an AS consists of information used to guide the AS through operand acquisition, the handling of requested operands by other ASes, determining when execution is possible

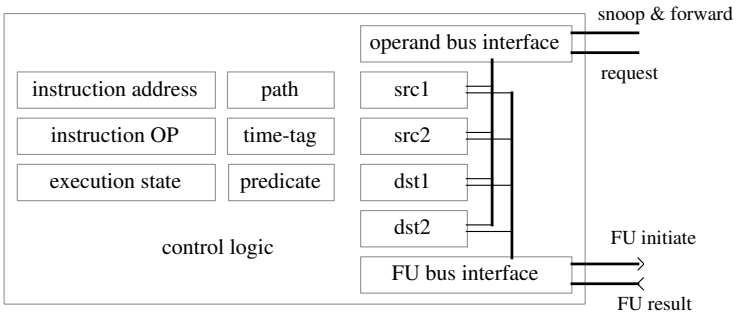

FIGURE 15.1: *High-level block diagram of the active station.* The major state associated with an active station is shown: four operand blocks (two source and two destination) and its four bus interfaces, grouped according to bus function into two major logic blocks.

or when re-executions are needed, and when commitment of this instruction is possible. Many state machines operate in parallel to manage the complex AS operational life-cycle and there is generally miscellaneous state associated with each.

A simplified block diagram of the active station is shown in Figure 15.1. 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. For example, 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 to accommodate dynamic predication since its state is included as part of the instruction-specific active station state mentioned previously.

Execution of the instruction can occur either entirely within the AS (like for the case of simple control-flow instructions and other simple instructions) or needs to be carried out with an execution unit or some sort. Most instructions generally require the use of an execution unit since the silicon size of the required execution pipeline is not only too large to fit within each AS but is something that benefits from being shared among many ASes. The complexity of the instructions for the ISA being implemented would be used to determine what type of instructions might be able to be executed within the AS itself. The nature and degree of sharing of execution resources among ASes is an implementation variation.

When execution is needed by a function unit, either a first execution or a re-execution, the AS contends for the use of an execution resource and upon

grant of such it sends the specific operation to be performed along with the necessary input operands and an identifying tag indicating the originating AS to the execution unit. The execution proceeds over one or more pipeline stages and the results are sent back to the originating AS. The AS then contends for outbound operand forwarding bus bandwidth and sends its output operands to future program-ordered ASes for their operand snooping and acquisition (discussed more below). Whether an AS is allowed to initiate a subsequent execution request before an existing one is completed (allowing for multiple overlapped executions of the same instruction) is an implementation variation. This is a performance issue that is still being researched.

The conditions determining when an AS can retire (either commitment or abandonment) vary somewhat according to specific implementations, but the more restraining condition of an AS retiring in a committed state generally must meet several criteria. Some of these are :

- all program-ordered past ASes are ready to retire
- the present AS is predicated to be enabled
- the present AS has executed at least once
- all input data operands have been snooped at least once (to verify correct value predictions)
- all of the latest created output operands have been forwarded

Specific implementations may add some other conditions for retirement (either commitment or abandonment) but when a retirement does occur, the AS is free to accept another dispatched instruction in the next clock.

15.3.6 Register and Memory Operand Storage

Register and memory operands are similar enough that they share an identical operand block within the active 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 active 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. 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 (architected) predicate registers (like

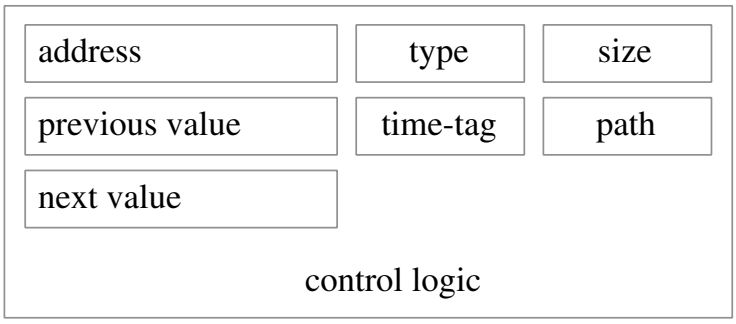

FIGURE 15.2: *Block diagram of an Operand Block.* Each Operand Block holds an effectively renamed operand within the active stations. Several operand blocks are employed within each active station depending on the needs of the ISA being implemented. The state information maintained for each operand is shown.

those in the iA-64 ISA [8, 18]. For memory operands, the identifying address is just the programmer-visible architected memory address of the corresponding memory value.

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, when the effects of the present instruction need to be squashed (if and when its enabling predicate becomes false), the previous value is broadcast forward. 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 15.2 shows a simplified block diagram of an operand block.

15.3.7 Operand Forwarding and Snooping

Similar to all microarchitectures, operands resulting from the execution of instructions are broadcast forward for use by waiting instructions. The time-tag and path ID accompany the operand's identifying address and value when it is forwarded. The time-tag will be used by subsequent instructions (later in program order time) already dispatched to determine 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 broadcast from one instruction to subsequent ones is referred to as a *transaction*, and generally consists of :

- transaction 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 transactions with the exception of predicate transactions (which generally contain more information). True flow dependencies are enforced through the continuous snooping of these transactions by each dispatched instruction residing in an issue slot that receives the transaction. Each instruction will snoop all operands that are broadcast to it but an operand forwarding interconnect fabric may be devised so that transactions are only sent to those instructions that primarily lie in future program-ordered time from the originating instruction. More information on operand forwarding interconnection networks is presented in a later section.

Figure 15.3 shows the registers inside an AS 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 *path* and *instr-time-tag* are only loaded when an instruction is dispatched.

If the path ID and the identifying address of the operand matches any of its current input operands, the instruction then checks 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 above evaluation is common for all operand snooping and provides for the dynamic discovery of feasible input operands.

15.3.8 Result Forwarding Buses and Operand Filtering

The basic requirement of the operand interconnection fabric is that it must be able to transport operand results from any instruction to those instructions in future program ordered time (those with higher valued time-tags). A variety of operand forwarding fabrics are possible but generally some combination of one or more buses have been the preferred choice so far. One choice is just to have one or more buses just directly interconnect all of the ASes,

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

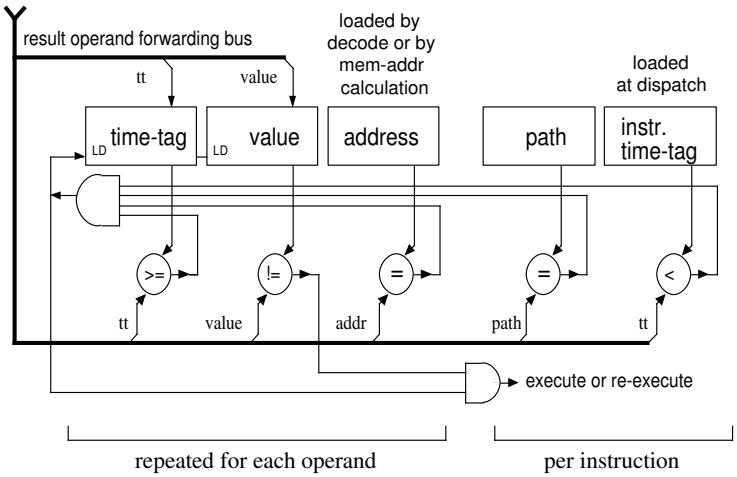

FIGURE 15.3: *Operand snooping.* The registers and snooping operation of one of several possible source operands is shown. Just one operand forwarding bus is shown being snooped but typically several operand forwarding buses are snooped simultaneously.

LSQ, and any register files (if needed). Although appropriate for smaller sized microarchitectures, this arrangement does not scale as well as some other alternatives. A more appropriate interconnection fabric that would allow for the physical scaling of the microarchitecture is one in which forwarding buses are segmented with active repeaters between the stages. Instructions lying close in future program-ordered time will get their operands quickly (since they are either on the same physical bus segment or only one or two hops away) while those instructions lying farther way will incur additional delays. This arrangement exploits the fact that register lifetimes are fairly short. [4, 20] Since these active repeaters forward operands from one bus segment to another, they have been termed *forwarding units*. A minimal forwarding unit would consist largely of a small sized FIFO. When the FIFO is full, the bus being snooped by that forwarding unit is stalled to prevent overflow. A variety of forwarding arrangements are possible: taking into account different buses, different operand types, and different forwarding units for each operand type.

In addition to allowing for physical scaling, the use of forwarding units makes possible a range of new microarchitectural ideas not easily accomplished without the use of time-tags and dynamic dependency determination. While some forwarding units can just provide a store-and-forward function, others can store recently forwarded operands and filter out (not forward) newly snooped operands that have previously been forwarded (termed *silent forwards*). This latter type of forwarding unit is therefore more specifically termed a *filter unit*. Filter units snoop and snarf operands in a way analogous to how ASes do but with slight differences to account for their forwarding

operation. Specifically, they also are assigned time-tag values identical to the ASes they lie next to in relation to the forwarding fabric. This filtering technique can be used to reduce the operand traffic on the forwarding fabric and thus reduce implementation costs for forwarding bandwidth. Two types of filter units have been explored so far. For registers, the *register filter unit* (RFU) can maintain either a small cache or a copy of all architecture registers within it and only forward those register operands whose values have not yet been seen by program-ordered future ASes. The ability to hold all of the architected registers also opens up the possibility for eliminating the need for a centralized register files (architected or not). Similarly, 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. The use of both types of these filter units is shown in the second of example microarchitecture presented in the next section.

15.4 Representative Microarchitectures

Two representative microarchitectures are now presented that both employ the Resource Flow execution model. The first is a relatively simple example that is oriented towards a machine with approximately the same physical silicon size (numbers of transistors) and complexity of a current or next-generation state-of-art processor. The second example shows how Resource Flow computing can be scaled to much larger physical sizes and numbers of machine resources. The possible machine size, in terms of resource components, with this example are currently substantially beyond those that can be achieved using any conventional microarchitectural models.

Both of these microarchitectures are similar to conventional microarchitectures in many respects. The common aspects of both microarchitectures are discussed first and then the distinctives of each is presented in subsequent sections. 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 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

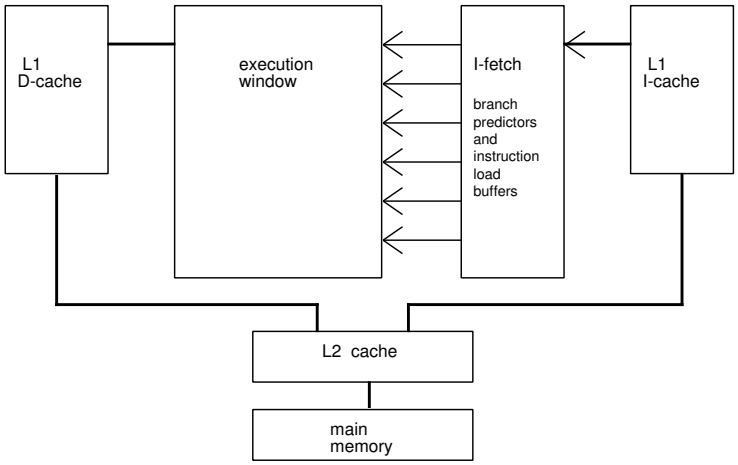

FIGURE 15.4: *High-level view of a Resource Flow microarchitecture.* Shown are the major hardware components of the microarchitecture. With the exception of the execution window block, this is similar to most conventional microarchitectures.

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 where the active 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 active stations within the execution window in a single clock.

Figure 15.4 provides a high-level view of both microarchitectures. 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. Instructions can be dispatched to active stations with or without initial input source operands. Instructions can also be dispatched with predicted input operands using value prediction. Once a new instruction is dispatched to an active station, once it has acquired its input operands (if it was not dispatched with predicted ones) it begins the process of contending for execution resources that are available to it (depending on microarchitectural implementation). Operand dependency determination is not done before either instruction dispatch (to an active 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 AS executing and possibly re-

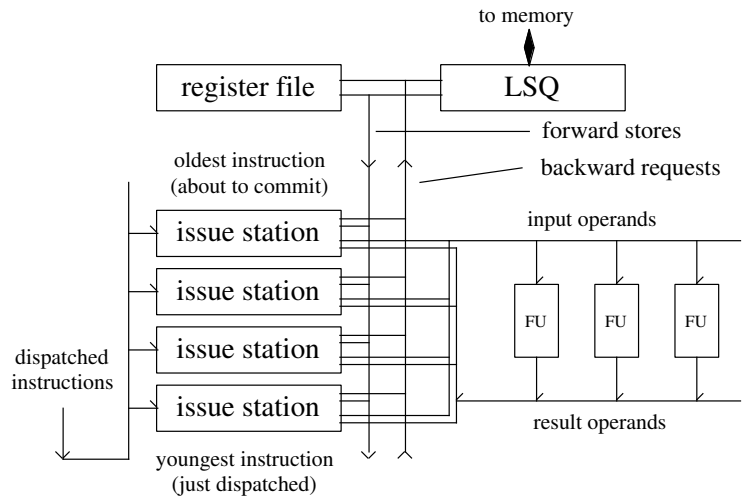

FIGURE 15.5: *High-level block diagram of a representative microarchitecture.* Active stations are shown on the left and various function units on the right. An architected register file and a load-store-queue is shown at the top. Bidirectional operand request and forwarding buses are shown vertically oriented (to the right of the active 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.

executing until they are ready to be retired. All retirement occurs in-order.

15.4.1 A Small Resource-Flow Microarchitecture

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

Figure 15.5 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 active 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 generally has independent pipeline depths. Instructions not executed by a function unit are executed within the active station itself. This currently includes all control-flow change instructions as well as load-store instructions but this is largely ISA dependent. The lower left shows (laid out vertically) the set of active stations (four are shown in this example) that may be configured into an implementation of the machine. The ASes 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.

In the center of Figure 15.5, running vertically, are two buses shown. These bidirectional and multi-master buses form the means to request and forward operands among the ASes, 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. Other bus arrangements are possible (some fairly complicated by comparison). In the present case, all buses can carry all operand types, however separate bus fabrics for handling different types of operands is also possible. One of these buses is used by ASes 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 ASes 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 ASes both receive and transmit on their connections to that bus.

Finally, unidirectional buses are provided to interconnect the ASes with the function units. One bus serves to bring instruction operations along with their source operands from an issue station to a function unit. The other bus returns function unit results back to its originating active station. Again these buses 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. The number of ASes in any given machine implementation roughly corresponds to the number of elements of a reorder buffer or a register update unit in a more conventional machine. 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,

the second representative microarchitecture introduced next is designed to explicitly address the issue of physical machine scalability.

15.4.2 A Distributed Scalable Resource-Flow Microarchitecture

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.

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 [3, 25] 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* 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. [26] Early forms of this microarchitecture were first presented by Uht [28] and then Morano. [13] A more detailed explanation of this microarchitecture can be found in [27]. Figure 15.6 shows a more detailed view of the execution window of this microarchitecture and its subcomponents.

Several ASes may share the use of one or more execution units. The execution units that are dispersed among the ASes 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 scalability is presented below), the ASes are laid out in silicon on a two-dimensional grid whereby sequentially dispatched instructions will go to sequential ASes down a column of the two-dimensional grid of ASes. The use of a two-dimensional grid allows for a design implementation in either a single silicon IC or through several suitable ICs on a multi-chip module. The number of ASes 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 ASes in a single clock.

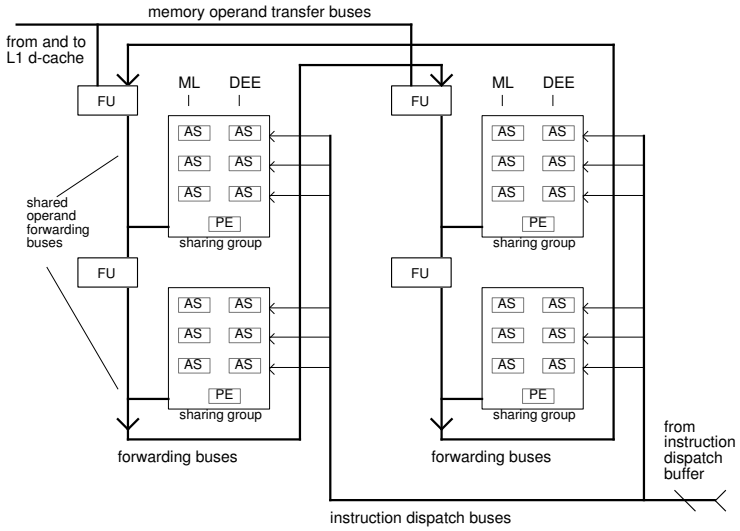

FIGURE 15.6: *The Execution Window of a distributed microarchitecture.* Shown is a layout of the Active Stations (AS) and Processing Elements (PE) along with some bus interconnections to implement a large, distributed microarchitecture. Groups of ASes share a PE; a group is called a *sharing group*.

Groups of active stations, along with their associated PE, are called a *sharing group* (SG), since they share execution resources with the set of ASes in the group. Sharing groups somewhat resemble the relationship between the register file, reorder buffer, reservation stations, and function units of more conventional microarchitectures. They have a relatively high degree of bus interconnectivity amongst them, as conventional microarchitectures do. The transfer of a decoded instruction, along with its associated operands, from an AS 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. Basically, only operand results need to flow from one SG to subsequent ones. As part of the ability to support multipath execution, there are two columns of ASes within each SG. The first AS-column is reserved for the mainline path of the program and is labeled *ML* in the figure. The second column of ASes is reserved for the possible execution of a DEE path and is labeled *DEE* in the figure. Other arrangements for handling multipath execution are possible but have not been explored.

The example machine of Figure 15.6 has a column height of six, and is shown with six instruction load buses feeding up to six ASes in the height dimension. Each SG contains three rows of ASes (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 4-tuple:

- sharing group rows
- active station rows per sharing group
- sharing group columns
- number of DEE paths allowed

These four characteristic parameters of a given machine are greatly influential to its performance, as expected, and the 4-tuple is termed the *geometry* of the machine. These four numbers are usually concatenated so that the geometry of the machine in Figure 15.6 would be abbreviated *2-3-2-2*. The set of mainline AS columns and DEE AS columns should be viewed as two sets of columns (one overlaid onto the other) that are really only sharing common execution resources and 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 buses to dispatch instructions from the i-fetch unit to the ASes and the more pervasive bus fabric to forward operands from ASes to program-ordered future ASes. The bus fabric and the units (labeled FU in the figure) are discussed in more detail below.

When an entire column of mainline ASes is free to accept new instructions, up to an entire column worth of instructions are dispatched in a single clock to the free column of ASes within that SG column. Newly dispatched instructions always to the column of ASes 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 ASes 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 ASes 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 AS. Again, instructions remain with their AS during execution and possible re-execution until they are ready to be retired. Unlike the previous example microarchitecture, an entire column of ASes gets retired at once (not at the granularity of a single AS). As a column of ASes 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 ASes within an AS column (either a mainline column or a DEE column) are assigned a contiguous set of time-tag values. The AS at the top of the column has the lowest valued time-tag within it, and the value is

increment by one for successive ASes down the column. The topmost AS of an adjacent AS mainline column will contain a time-tag value one higher than the previous column's AS having the highest valued time-tag (which will be located at the bottom of the previous column). The time-tag value assigned to ASes within a row and jumping from one SG column to an adjacent one (one mainline column of ASes to another in an adjacent SG column) have time-tag values differing by the height of a column as counted in ASes. At any one point in the machine operation, one column serves as the *leading* column with the AS having the lowest valued time-tag (generally zero) within it, while another column serves as the *trailing* column with the AS having the highest valued time-tag. The lead AS column contains those instructions that were dispatched earliest while the trailing AS column contains those instructions most recently dispatched.

It will be the leading mainline AS column that is always the one to be retired next. Upon a retirement of that column, the time-tags within ASes and operands within the machine are effectively decremented by an amount equal to the height of the machine in ASes (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 AS 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 ASes 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.

15.4.2.1 Bus Interconnect and Machine Scalability

As with the previous example microarchitecture, some sort of interconnection fabric is used to allow for the backward requesting of operands among ASes as well as for the forwarding of result operands. Generally, several buses are used in parallel to handle the operand forwarding bandwidth requirement.

All operand buses are multiplexed for use by all operand types. Multiplexing of all operands onto a single set of buses provides the best bandwidth efficiency but represents just one possible arrangement. The set of operand forwarding buses, partitioned into segments and separated by forwarding units (FUs in the figure), are shown vertically (primarily) and to left of of each SG column. For clarity, the backwarding operand request buses are not shown in the figure.

In order to make the machine scalable to large sizes, all operand buses are of a fixed (constant) length, independent of the size of the machine (in numbers of components, ASes and PEs). This is accomplished through the use of forwarding units (previously discussed). This present microarchitecture actually uses three different types of forwarding units (one each for predicates, registers, and memory operands) but they are all shown combined in Figure 15.6 as the boxes labeled FU. Note how the operand forwarding bus segment bundles loop around from the bottom of one SG column to the top of the adjacent SG column to its right, with the forwarding bus bundle from the bottom of the rightmost SG column looped around to the top of the leftmost SG column. This is a logical arrangement only and does not necessarily represent how all components might be laid out in silicon. This looping arrangement for operand forwarding follows the familiar pattern characteristic of many proposed larger-scaled microarchitectures. [16]

For the register and memory operands, the forwarding units also provide filtering and are therefore RFUs and MFUs respectively. The register filter units (RFUs) used here contain a set of registers equal in number to that in the set of architected registers for the ISA being implemented. This provides a sort of 100capability for registers. Of course, this may only be possible when the set of architected registers in the ISA (including status registers and any explicit predicate registers is rather small – maybe less than 512 or 1024). Requests by ASes for register operands therefore will always "hit" in the preceding RFU (in program-ordered time) unless the register being requested has been invalidated for some reason due to the forwarding strategy employed. Generally different forwarding strategies are used for register, memory, and predicate operands but full details on this are not the present interest. Likewise, the memory filter units employ an internal L0 cache so that requests for memory operands have a chance of hitting in the MFU situated immediately preceding the AS making the request. Operand hits in either the RFUs or the MFUs also serve to reduce operand bus traffic. Operand misses (either RFU or MFU) are propagated backwards until a hit is achieved. For registers, a hit will always eventually be achieved even if the request has to travel back to that RFU with the time-tag value of zero (which always has a valid value). Similarly for memory requests, misses propagate backwards, however a hit is not always guaranteed as with registers. A miss that travels back through that MFU that has a time-tag value of zero continues backward to the L1 data cache, and possibly on up to higher memory levels as well (the L0 inside the MFU being the lowest, L1 being higher, et cetera). A memory

request that misses in an MFU at the top of an SG column, switches from using the backwarding bus fabric to the bidirectional and multimaster *memory operand transfer buses* (also a duplicated bundle of buses) shown at the top of the figure. Memory requests returning from the L1 data cache enter an MFU at the top of the machine and get forwarded normally on the forwarding fabric until snarfed by a requesting AS. For this microarchitecture, MFUs also replace the LSQ function in more conventional microarchitectures.

15.4.2.2 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. [17]

15.4.2.3 Persistent Architected Register State

Persistent register, predicate, and some persistent memory state is stored in the forwarding units. Persistent state is not stored indefinitely in any single forwarding unit but is rather stored in different units as the machine executes column shift operations (columns of ASes get retired). However, this is all quite invisible to the ISA. This microarchitecture also implements precise exceptions [19] 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 AS that contains the generating instruction until it would be committed. No action is needed for pending exceptions in ASes that eventually get abandoned (squashed). When an AS with a pending exception does commit, the machine directs the architected control flow off to an exception handler through the defined exception behavior for the given ISA.

15.4.2.4 Multipath Execution

DEE paths are created by dispatching instructions to a free column of ASes that is designated for holding DEE paths. 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 ASes) 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. A more detailed and thorough discussion of multipath execution and disjoint eager execution was covered in Chapter 6.

15.5 Summary

A set of concepts and some basic microarchitectural components, along with an unrestrained model of speculative execution that has been termed Resource Flow computing, has been briefly presented. Also briefly presented was the design outline for two example microarchitectures where each embodies many of the Resource Flow ideas (although each to a different degree). These microarchitectures show how this model of computing might be implemented both for more modest sized machines (as compared with the next generation processors) and future large sized machines having approximately one billion or more transistors. Using the silicon technology design rules that were used for the Alpha EV8 processor, the second microarchitecture presented was estimated to have a transistor budget of approximately 600 million transistors for an 8-4-8-8 geometry machine configuration (256 ASes, issue width and number of PEs of 64, instruction dispatch and commit width of 32). Simulated performance of this configuration using existing memory performance specifications achieved a harmonic mean IPC across a mix of SpecInt benchmarks of about 4. Whether this level of performance is worth the amount of machine resources (as compared to other simpler alternatives) remains to be seen, but research is ongoing.

The basic Resource Flow technique as well as both microarchitectures presented can be applied to existing legacy ISAs (usually an important requirement in the commercial market). The design ideas presented are not oriented towards all future processor needs but rather for those application needs where the maximum performance is required for a single thread of control.

References

[1] 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, Jan. 1995.

[2] Farkas K.I., Chow P., Jouppi N.P., Vranesic Z. The multicluster architecture: Reducing cycle time through partiioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, 1997.

[3] Ferrante J., Ottenstein K., Warren J. The progream dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] 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*, pages 236–245, New York, NY, Dec 1992. ACM Press.

[5] Gonzalez J. and Gonzalez A. Limits on Instruction-Level Parallelism with Data Speculation. Technical Report UPC-DAC-1997-34, UPC, Barcelona Spain, 1997.

[6] Hinton G., Sager D., Upton M., Boggs D. Carmean D., Kyker A. Roussel P. The microarchitecture of the Pentium-4 processor. *Intel Technical Journal - Q1*, 2001.

[7] Intel. *Intel Itanium2 Processor Reference Manual for Software Development and Optimization*, June 2002.

[8] Intel Corp. *iA-64 Application Developer's Architecture Guide*, 1999.

[9] Kemp G.A., Franklin M. PEWs: A decentralized dynamic scheduler for ILP processing. In *Proceedings of the 24th International Conference on Parallel Computing*, pages 239–246, 1996.

[10] Lam M.S. and Wilson R.P. Limits of Control Flow on Parallelism. In *Proc. of ISCA-19*, pages 46–57. ACM, May 1992.

[11] Lipasti M.H and Shen J.P. Superspeculative Microarchitecture for Beyond AD 2000. *IEEE Computer*, 30(9), Sep 1997.

[12] Morano D.A. Execution-time Instruction Predication. Technical Report TR 032002-0100, Dept. of ECE, URI, Mar 2002.

[13] 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 MEDEA Workshop (held in conjuction with PACT'02)*, 2002.

[14] 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*, New York, NY, Nov 2001. ACM Press.

[15] 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 simultane-

0-8493-0052-5/00/$0.00+$.50

ous multithreading. In *Proceedings of the International Solid State Circuits Conference*, Jan 2002.

[16] 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*, pages 272–281, New York, NY, October 1998. ACM Press.

[17] Sankaranarayanan K. and Skadron K. A Scheme for Selective Squash and Re-issue for Single-Sided Branch Hammocks. IEEE Press, Oct 2001.

[18] Schlansker M.S. and Rau B.R. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, Feb 2000.

[19] Smith J.E., Pleszkun A.R. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, Sep 1988.

[20] Sohi G.S., Breach S. and Vijaykumar T.N. Multiscalar Processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, New York, NY, Jun 1995. ACM Press.

[21] Sundararaman K.K., Franklin M. Multiscalar execution along a single flow of control. In *Proceedings of the International Conference on Parallel Computing*, pages 106–113, 1997.

[22] Tjaden G.S., Flynn M.J. Representation of concurrency with ordering matrices. In *Proceedings of COMPCON*, volume C-22, pages 752–761, Aug 1973.

[23] Tomasulo R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[24] Tsai J-Y., Yew P-C. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, 1996.

[25] Uht, A. K. An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code. In *Proceedings of the Nineteenth 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.

[26] Uht A. K. and Sindagi V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. MICRO-28*, pages 313–325. ACM, Nov 1995.

[27] Uht A.K., Morano D., Khalafi A., Alba M. and Kaeli D. Levo – A Scalable Processor With High IPC. *Journal of Instruction Level Parallelism*, 5, Aug 2003.

[28] 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.