A Distributed Microarchitecture for Realizing High-ILP

A. Khala£, D. Morano, D.R. Kaeli
Northeastern University
akhala£, dmorano, kaeli@ece.neu.edu

A.K. Uht
University of Rhode Island
uht@ele.uri.edu

***Abstract*— Many existing programs, and applications in general, are very serial in nature and cannot easily be parallelized for gaining execution speed-ups. Existing, conventional, microarchitectures have not been able to extract large amounts of ILP from these sorts of programs. Rather, new, large, and more aggressive out-of-order microarchitectures are needed for this task. We introduce a new microarchitecture suitable for extracting large amounts of instruction level parallelism (ILP) from sequential oriented programs. Instructions per clocks (IPC) in the range of 3.8 to 5.1 are obtained for the range of machine con£gurations studied running SpecInt-2000 and SpecInt-95 programs.**

I. Introduction

WE introduce a new distributed microarchitecture for achieving a high amount of instructions per clock (IPC) from general sequential program codes. Lam et al [10] showed that a large amount of instruction level parallelism (ILP) is present within typical program codes. Results by Uht et al [22] con£rmed this and showed in more detail how minimal control dependencies and multipath execution are important in exploiting ILP. Gozalez et al [5], [6] also showed the potential for ILP when considering value prediction. Suitable microarchitectures for extracting this ILP would need to execute a very large number of instructions in parallel. Unfortunately, conventional microarchitectures currently suffer from hardware resource contention and routing congestion problems when scaled to large sizes [16]. Particularly severe contention occurs for microarchitectures using issue windows, register update units, or reorder buffers [19], [1], [8]. Microarchitectures that have been proposed to attempt to mitigate the problems associated with large numbers of parallel execution resources have either relied, in part, on assistance from a compiler, thus rendering them unsuitable for legacy instruction set architectures (ISA). Microarchitectures such as the Multiscalar processor [20] and the Grid Processor Architecture [15] have been of this type.

Many microarchitectural ideas appear to offer potential for exploiting ILP beyond what conventional designs can do. Some of these ideas include multipath execution, aggressive control-¤ow speculation, value prediction, and memory handling through aggressive memory disambiguation. Some microarchitectures capable of multipath execution include those described by Wang [25], Uht and Sindagi [22], Heil and Smith [7], the PolyPath by Klauser et al [9], and a simultaneous multithreading oriented approach by Wallace et al [24]. Microarchitectures that attempt to execute control-independent instructions beyond the join of conditional branches (minimal control dependencies) are described by Sankaranarayana and Skadron [17], [18], Cher and Vijaykumar [2], and Uht et al [23]. Employing value speculation [11], [12] also appears to be a signi£cant advance in exploiting inherent ILP by breaking the data dependence graph of the program. A representative microarchitecture of this sort would be the Superspeculative machine of Lipasti et al [13]. Aggressive memory handling for large microarchitectures was also proposed by Franklin et al [4]. And a promising mechanism for managing the problem of program execution result ordering was employed in the Warp Engine [3].

Our goal was to attempt to combine the best features of both existing microarchitectures and microarchitectural proposals, along with new ways to handle problems associated with existing designs. We drew from many of the above ideas and propose a space-distributed, scalable, grid-oriented microarchitecture that features: multipath execution, minimal control dependencies, value prediction, aggressive memory speculation and handling, and time tags for maintaining program dependency order.

The rest of this paper is organized as follows. Section 2 introduces a new model for guiding program execution and the management of machine hardware resources during execution. This new execution model is termed *Resource Flow Execution*. Section 3 provides the description of a microarchitecture that realizes our Resource Flow execution model. Section 4 provides an experimental evaluation of the effectiveness of the described microarchitecture. We summarize in Section 5.

II. Resource-Flow execution model

One of our motivations for the design of a new microarchitecture was to exploit as much speculation as possible. We believe that achieving high instructions per second (IPC) requires going beyond the traditional data and control ¤ow dependency constraints in executing a program to allow for more parallel execution of the code. In current microarchitectures that perform speculative execution, access to the reorder buffer becomes very problematic as the number of instructions being speculatively executed concurrently grows [16]. This is mainly due to the contention for the centralized resources such as data registers within the reorder buffer. In the proposed resource ¤ow execution model the instructions with the highest priority that are not yet executed are assigned to free execution resources. The assignment is done regardless of whether the input operands of the instructions have the correct value (data ¤ow constraints) and regardless of the outcome of the previous branches in the instruction stream (control ¤ow constraints). This means that the execution is guided based upon the need or desire for an instruction to execute and the availability of resources. The rest of the execution time is spent re-executing the instructions that have had an incorrect speculative input or operand, so as to end up with a programmatically correct execution of the code. As demonstrated in the rest of this section, through application ??? of the Resource-Flow execution model there is no need for explicit renaming of registers or any reorder buffers that would eliminate the contention for centralized resources. This model provides methods for executing standard procedural (control-¤ow guided) programs that go beyond the standard control ¤ow model and the data ¤ow model.

A. Time Tags

In the Resource-Flow execution model, instructions are executed speculatively in an out-of-order fashion. We use time-tags to enforce the correct data and control dependencies among the instructions to realize speculative data-¤ow execution of code. A time-tag is a small integer that uniquely identi£es the position of an instruction or an operand in program ordered time with respect to the most recently retired instructions. Time-tags, as used in our microarchitecture were originally proposed in the Warp Engine [3]. This machine relied on the use of ¤oating point numbers for the time-tags. In our microarchitecture, time-tag values refer to the issue slots in the execution window and are small integer values.

To assign the time-tags, the oldest issued instruction in ¤ight that is neither yet committed nor squashed would usually have a time-tag value of zero. More recently dispatched instructions take on increasingly larger values the further ahead they are in the program ordered instruction stream. As a group of instructions with the lowest valued time-tags are retired, all of the time-tags for all instructions and operands are decremented by the number of instructions just retired. This will keep the next instruction to be retired in having the time-tag value of zero and therefore prevents time-tags from growing inde£nitely.

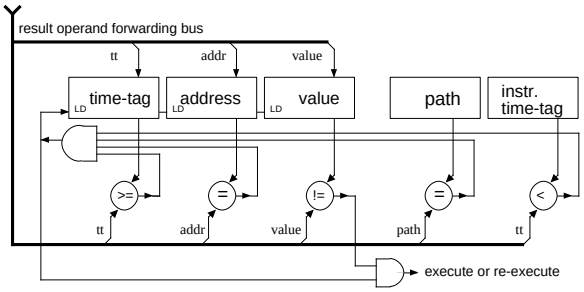


Fig. 1. Operand snooping logic within an AS.

Both renaming and time-tagging assume *forwarding* of instruction result operands on a bus, snooped by all later instructions. In the resource ¤ow model, the forwarding of the identifying address of the output operand and its value is accompanied with the instruction time-tag and path ID (if multipath execution is allowed) ??? of the forwarded result. The time-tag and path ID will be used by subsequent instructions in the execution window to determine if the operand has the desired address value and therefore should be snarfed as an input that will trigger its execution or re-execution. Our path ID is similar to that used by Skadron et al /citeSkadron01.

The issue slots in our microarchitecture are modi£ed to handle instruction execution, forwarding of operand results, and snooping of instruction results. We call this modi£ed issue slot an Active Station (AS). Figure 1 shows the detailed snooping hardware for an input operand of an Active Station. Basically, a new operand is snarfed when it has the same address and path identi£er (if multipath execution is allowed) ??? as the current AS, as well as a time-tag value that is less than that of the current AS itself but greater or equal to that of the last snarfed operand.

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
- pperand Type
- path ID
- time-tag of the originating instruction

inst.. Label	Time Tag	Inst. Mnemonic	
I1	1	lui	r3,0x8002
…			
I3	3	lw	r8,-29(r28)
I4	4	addiu	r5,r8,16
I5	5	lw	r3,-26(r28)
…			
I7	7	addiu	r5,r1,32
I8	8	xor	r8,r3,r5
I9	9	sw	r8,-32(r28)

(a) Code fragment

Cycle	Execute	Forward	Snarf
-1		Ix(r28),Iy(r1)	I3(r28),I5(r28),I7(r1),I9(r28)
+0	I1,I3,I5,I7		
+1		I1(r3),I7(r5)	I8(r3,r5)
+2	I8	I3(r8),I5(r3)	I4(r8),I8(r3),I9(r8)
+3	I4,I8,I9		
+4		I4(r5),I8(r8)	I9(r8)
+5	I9		

(b) Execution schedule

Fig. 2. Example instruction execution.

- identifying address
- data value for this operand

True ¤ow dependencies are enforced through continuous snooping of these transactions by each dispatched instruction residing in an active station that receives the transaction.

Figure 2 is an example of how time-tags are used to enforce correct execution of instructions by enforcing data dependencies. Part (a) shows a fragment of a MIPS code. For simplicity, in this example we assume that there are no branches among these instructions and that they are all eventually committed. Each instruction is assigned to an active station with a unique time-tag.

Figure 2(b) illustrates the steps required to execute this code fragment. For simplicity, we assume that there is no constraint on the number of execution units and the number of transactions that can be forwarded in each cycle. The £rst column shows the relative clock cycle times. The next three columns list the instructions that are either executing, forwarding an operand, or snar£ng a snooped operand value. The notation that is used is $I_x(r_y)$ where I_x is the instruction label and r_y is the register that is either forwarded or snarfed. Note that snar£ng is done in parallel with instruction execution and transaction forwarding.

At clock 0 instructions I_1, I_3, I_5 and I_7 are executed in parallel. The execution is the result of the snar£ng of $r1$ and $r28$ architected registers in the previous cycle. Assuming two cycles for the execution of load and store instructions, and one cycle for the rest of the instructions in this example, instruction I_1 and I_7 will produce their results in the next clock. The new value for registers $r3$ and $r5$ are forwarded in clock 1 and are snarfed by instruction I_8. In the next clock, instruction I_8 will be executed using its newly read register value. Normally I_8 should forward the new result, but since I_5 is sending out a new value for $r3$, the I_8 instruction snarfs the new value of $r3$. This results in re-execution of I_8 instruction at clock +3. In the next cycle, I_4 sends a new value for register $r5$ with a time-tag of 4, but since the last value of $r5$ received by I_8 had a time-tag of 7, which is greater than 4, the new value is ignored by I_8 and does not result in another re-execution.

B. Memory operations

With the increasing number of instruction in ¤ight, there is a higher probability that the memory values generated by store operations will be used by other load operations in the execution window. Table XXXX shows the ratio of memory operations that can get their operand values directly from other instruction in the execution window without having to go to a higher level of memory hierarchy. This means that if we could somehow satisfy a portion of the memory operations in the execution window, it would result in less pressure on the higher levels of memory hierarchy and overall performance improvement.

To enforce the true memory dependencies, we could still use the forwarding and snooping mechanism introduced in the last section. There are, however, limitations to the applicability of this approach. Unlike with register operands, the architected address for a memory operand is not £xed. The memory address is generally computed using a £xed displacement and a register value. Due to the speculative nature of this microarchitecture, the register value for the memory operation could be incorrect, resulting in a wrong memory address.

A load operation that snoops for an incorrect memory address will miss the opportunity to snarf the correct memory value forwarded by a previous store operation. It is therefore necessary for a load operation to eventually initiate a memory request using its resolved memory address. In traditional microarchitectures, the request normally goes to the £rst level D-cache. In our proposed microarchitecture, there is a good chance that the memory value is still in the execution window and has not yet been written back to the higher level memory. Based on

this observation, we send memory requests to both higher level memory and previous active stations. If there is any store operation with the same address in earlier active stations, it will forward the memory value, which will be used subsequently by the load instruction. Otherwise the memory value received form the higher level memory is used. The requests to the higher level memory are sent using a set of shared horizontal buses. These are termed *horizontal* since they do not follow the vertically oriented serpentine buses shown in Figure 5. Rather, these horizontal buses are pictured in Figure 6. The requests to the previous active stations are handled using a set of *backwarding* buses. These buses are essentially similar to the forwarding buses except that they are primarily used to send data requests to the instructions earlier in the program order. More discussion on memory structure and busing will be presented in later sections.

Another dif£culty with the application of the forwarding and snooping strategy for memory operations is that a store operation with an incorrect memory address could forward an erroneous value that will be incorrectly snarfed by another load operation with the same address. This presents a problem for the correct enforcement of memory operand dependencies. We solved this by forcing the store operation to notify subsequent load instructions to ignore its previously forwarded value. We de£ne *nullify* transaction as a new type of transaction with the property of canceling the effect of a previous store transaction. Any store instruction that has previously forwarded a memory value with the incorrect speculative address, will send a nullify transaction upon re-execution. This transaction is seen by all later load instructions. Any load instruction that has snarfed a value sent from that speci£c store instruction will ignore the value and sends a new request to the memory and on the backwarding buses.

C. Hardware Predication

Predicated execution is shown to be an effective approach for handling conditional branches []. In our microarchitecture, the predicates are generated at run-time. Each instruction computes its own enabling predicate by snooping for and snar£ng predicate operands that are forwarded to it by earlier instructions from the program-ordered past. They are evaluated solely with hardware, allowing the use of legacy code. Full hardware based predication is a new way to manage and achieve minimal control dependencies (MCD). With MCD, all branches may execute concurrently, and the instructions after a branch domain [21] may execute independently of the branch.

In this section we propose a hardware predication mechanism that can be implemented using our resource

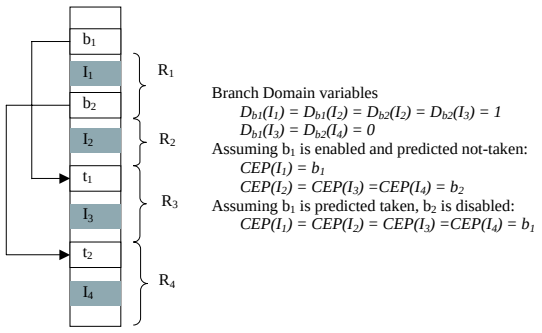
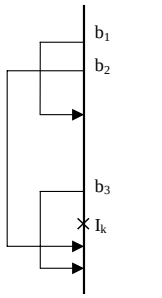


Fig. 3. Branch Domains and Closest Enabled Previous (CEP) Branch.



(a)

	Branch Status			Inst. Status		Transactions	
Cycle	b_1	b_2	b_3	I_k	CEP	Pred.	Inval.
0	T	D	N	E	b_1		
+1	N	D	N	E	b_3	b_1.nt	
+2	N	T	N	E	b_3	b_2.t	
+3	N	T	D	E	b_3		$b_3(b_2)$
+4	N	T	D	D	b_2		
+10	T	T	D	D	b_2	b_1.t	
+11	T	D	D	D	b_2		$b_2(b_1)$
+12	T	D	N	D	b_2	b_3.nt	
+13	T	D	N	E	b_3		

(b)

Fig. 4. Example of how Dynamic Predicate works.

¤ow methodology. Figure 3 shows a program code sequence with two branches b_1 and b_2. with corresponding targets t_1 and t_2. The two branches divide the code into 4 regions. The execution of the instructions in regions R_1, R_2 and R_3 depends on the outcome of the b_1 and b_2 whereas the instructions in R_4 are executed independent of the branch outcomes. In this section, we are proposing a new scheme for assigning an enabling predicate to every instruction in each region based on the outcome of branches. If the enabling predicate value is one, then the instruction will be executed; otherwise it is disabled. The following de£nitions are used in the description of our scheme.

T_b:	a binary value, set to one if the branch b is predicted taken
ex_j:	a binary value assigned to each instruction I_j and speci£es whether the instruction is executed or otherwise disabled.
$D_b(I_j)$:	a binary value, set to one if the instructions j is in the domain of branch b.
$CEP(I_j)$:	Closest Enabled Previous branch to instruction I_j in the static program-order.

Figure 3(b) shows an example along with the value of

some the variables de£ned above. The $D_b(I)$ function is independent of the outcome of the branch as only depends on the static order of instructions in the code. $CEP(I)$, on the other hand, is a function of the outcome of other branches and will change during the course of speculative execution.

Using the above de£nitions, we can see that

$\overline{ex_j} = T_b \cdot D_b(I_j)$ where $b = CEP(I_j)$. (1)

This equation simply tells us that if an instruction I_j is in the domain of an enabled branch, and the branch is taken, it will not be executed. If the branch is out of the domain, then its execution is independent of the outcome of the branch.

Equation (1) is the base for our dynamic predication scheme. To set the predication for each instruction I_j, it is suf£cient to £nd the $CEP(I_j)$ branch and its outcome. This is a simple task in our scheme. Each active station has a pair of CEP_TT registers to keep the time-tag of its current CEP branch and corresponding target. If an earlier branch is enabled, a *Predicate* transaction is sent on the forwarding bus with the branch time-tag, its target address and whether it is a taken branch or not. Each subsequent instruction that snoops this transaction checks to see if the time-tag of the newly enabled branch is greater than its current CEP_TT value. If so, the new branch is closer to this instruction and therefore it will replace the older CEP branch.

If a branch such as b is disabled, the later instructions need to obtain the new CEP branch. The new branch is simply $CEP(b)$ because there is no other enabled branch between it and b. In our scheme this is handled by forwarding an invalidate transaction by branch b. An invalidate transaction contains the time-tag of b, the time-tag of $CEP(b)$ and the branch domain information of $CEP(b)$. This information was stored at b when $CEP(b)$ forwarded its predicate transaction.

To £nd the domain of a CEP branch, a simple approach is to just compare the branch target address with the instruction address. It is a however better to use time-tags for this purpose. To do so, a branch needs to forward the time-tag of its target instruction instead of its target address. The branch target time-tag can be easily calculated by adding the branch displacement to the branch time-tag value.

Another improvement to the above scheme can be made by noticing that a not-taken enabled branch does not need to send out any transaction on the forwarding bus. This is due to the fact that a not taken branch essentially acts like a NOP operation. The advantage is a reduction of the number of transactions on the bus. Figure 4 shows an example of how our dynamic predicate work. The £rst column in the table lists the relative clock cycles. The next three columns list the status of each branch. The "D", "T' and "NT" entries correspond to disabled, taken and not-taken status respectively. The next two columns list the I_k Instruction status. "E" and "D" stand for enabled and disabled status respectively. The CEP column shows the $CEP(I_k)$ branch at any cycle. The next two columns show the transactions on the bus. The "pred." column lists the predicate forward transactions for each branch along with its status. The "inval." column lists the invalidating transactions. the branch name in the parenthesis corresponds to the new CEP forwarded by the invalidated branch.

In the example, it is assumed that b_1 and b_3 are initially predicted taken and not-taken respectively and a as a result I_k is enabled. In the next clock, b_1 changes its direction to a not-taken branch and sends a forwarding transaction on the bus. This transaction will enable b_2 which is also predicted taken. b_2 will send a predication transaction on the forwarding bus which is snooped by b_3. As a result, b_3 will be disabled in clock 3 and will send an invalidating transaction with b_2 as its CEP branch. instruction I_k will see this transaction and switch its CEP to b_2 and as a results will be disabled. The bottom section of the table in £gure xx shows what will happen if b_1 changes back to a taken status. A new set of transactions will follow that eventually enable I_k.

In the previous example we saw that I_k was £rst disabled and then enabled. A disabled instruction needs to notify *later* instructions that its destination register value, which could have been forwarded earlier, is not valid anymore. Further, it has to send the old destination value from the closest enabled previous instruction as the correct value. This is done through a relay transaction where the previous value is sent out as the new value produced by this instruction. The disabled instruction, in effect acts like a relay and will forward the earlier value using its own time-tag.

As demonstrated by this example, the cost of hardware predication is low, since most of the extra state storage only takes a few bits in the AS. This hardware stays same for all active stations and columns

III. A Resource Flow Microarchitecture

In this section we present an ISA independent, scalable, distributed microarchitecture that uses Resource Flow execution model and some of the most desirable features of future microarchitectures in order to achieve large IPC. Among these are control ¤ow speculation, control-independent and multipath execution. The microarchitec-

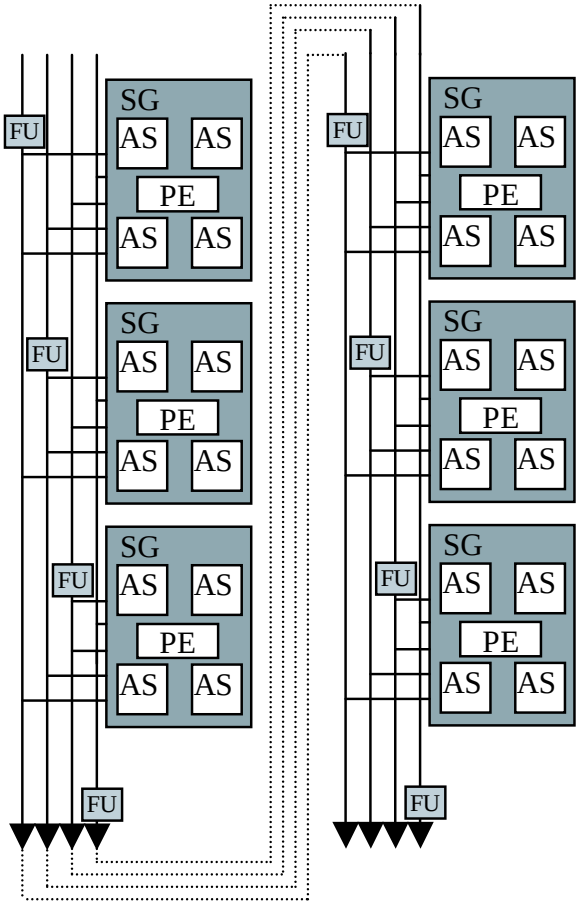


Fig. 5. Interconnection fabric in the Execution Window.

ture 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 such as register £le, reorder buffer and centralized execution units are eliminated through distribution of theses resources.

Figure 5 shows a partial diagram of our execution window with its subcomponents. The active stations are laid out in a two dimensional grid where sequentially loaded instructions go to sequential active stations down a column. The use of a two dimension grid simply provides a means to implement the necessary hardware in a chip. Dispersed among the active stations are associated execution units. An execution unit is represented in the £gure as processing element (PE). Processing elements may consist of a uni£ed all-purpose execution unit capable of executing any of the possible machine instructions or, more likely, consist of several functionally partitioned units individually tailored for speci£c classes of instructions (integer ALU, FP or other), as is typical of most current machines. Groups of active stations along with their associated processing elements are termed a sharing group (SG).

Sharing groups have a relatively high degree of bus interconnectivity within them, as conventional microarchitectures do. The Active stations serve the role of both the reservation station and the reorder buffer of more conventional machines. The transfer of a decoded instruction, along with its associated operands, from an AS to its PE is isolated to within the given SG. The use of this execution resource sharing arrangement also allows for reduced interconnections between adjacent sharing groups. Basically, only operand results need to ¤ow from one SG to subsequent ones.

A. Segmented Buses

An interconnection fabric is provided to forward result operands from earlier active stations to later active stations in program order. The interconnect allows for arbitrary numbers of sharing groups to be used in a machine while still keeping all bus spans to a constant length. All the buses in Figure 5 form the interconnection fabric. The width of the buses can be increased by using several buses in parallel which will increase the overall bus bandwidth.

Active bus repeater components are required to allow for constant length bus spans. Adjacent bus segments are connected via Filter Units (FU). These units do more than just repeat operand values from one span of a bus to the next. For registers and memory, operands are £ltered so that redundant forwards of the same value are eliminated. This £ltering provides a means to reduce the overall bandwidth requirements of the forwarding interconnection fabric. There are three kinds of £lter units. The Register Filter Unit (RFU), Memory Filter Unit (MFU) and Predicate Filter Unit (PFU). Each type handles its corresponding transactions. Each RFU holds the most recent value with the highest time-tag for each of the architected registers. The register values are then forwarded to the next bus segment during the course of execution. The MFU has basically the same structure of an RFU but it uses a small cache (L0 cache) to store the latest forwarded memory values. The L0 cache proves to be an effective device in reducing the number of requests to L1 D-cache. However the limited size of L0 cache requires periodic writes to the L1-D cache. Among our £ltering units, the PFU are the simplest ones. All they need to do is to keep track of the latest CEP branch and forward it to the next bus span.

Register £lter units result in the elimination of a cen-

tralized register £le and simpli£cation of state commitment. By the time that the instructions in a column have gone through their £nal execution and the column is ready to commit, the RFU's in this column have already forwarded the register values to the RFU's in the later columns. Therefore it is unnecessary to save the ISA register state in a separate register £le.

As can be seen in £gure 5, the buses go from the bottom of each column to the top of the next column. The bottom of the last column in the execution window is connected to the top of the £rst column. This will create a loop where there is not really a £rst or last column but there are *earliest loaded column* (ELC) and the *latest loaded column* (LLC). The operand values are forwarded from each column to the next column, except for the LLC column which will forward its operand values only when the ELC column is committed and loaded with a new set of instructions.

B. Multipath Execution

Control-¤ow misprediction penalties pose an obstacle for designing high performance microarchitectures. Both MCD and Disjoint Eager Execution (DEE) techniques are needed for achieving very high ILP [22]. DEE gives the highest priority for the use of execution resources to the instructions with the highest likelihood of being committed. In our microarchitecture, the likelihood is not explicitly calculated; instead the *"static tree"* heuristic of [22] is used. This is a form of multipath execution in which there is a predicted or mainline path as well as several much shorter not-predicted or *disjoint* paths spawning from the mainline path at some conditional branches.

DEE paths are created by loading a free second column of active stations within a SG column. Instructions from both mainline and disjoint paths share the same PE and bussing resources in each sharing group. Active stations on the Mainline path, however, always have priority for the use of resources. In our present microarchitecture, we always have two columns of active stations within a SG. The £rst AS column is reserved for the main-line path of the program. The second column of Active stations is reserved for the possible execution of a disjoint path. The cost impact of realizing DEE is however relatively low and is usually less than 10 percent increase for a 45 precent performance improvement.

Mainline and disjoint paths execute concurrently, greatly reducing branch misprediction penalties. Conditional branches are assigned to a free disjoint path (the path is *spawned*) after they enter the execution window. The column with the disjoint path could be any free column in the execution window and not necessarily the same column as their mainline path column. This allows for multiple disjoint paths for a single mainline path. At branch resolution, if the disjoint path turns out to be the correct path, it will replace the mainline path and the execution will continue form there. A more detailed description of DEE can be found in [22].

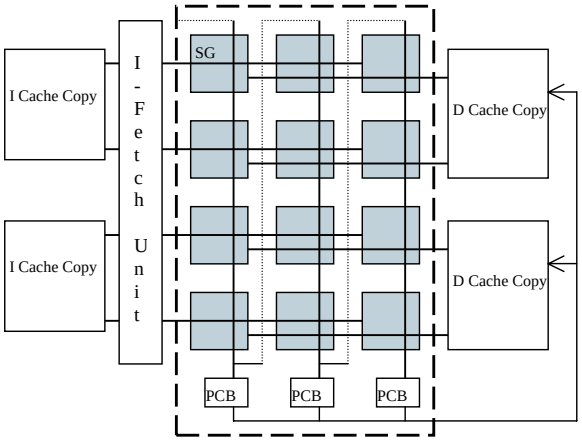


Fig. 6. High level diagram of our microarchitecture.

C. Instruction Fetch

The challenge in high bandwidth instruction fetch is mainly concerned with the problem of multiple-branch prediction and aligning and collapsing of multiple fetch groups. One approach for solving this problem has been focused on enhancing the instruction cache by using multi-ported, multi banked copies of the instruction cache[][][]. A more recent approach has been the use of trace caches[][].

A high level diagram of our microarchitecture is shown in £gure 6. We are using a set of replicated multiported I-caches to satisfy our high fetch bandwidth requirements. Instruction are loaded in the active stations using a set of buses that are not shown in the £gure. The key here is that instructions with small branch domain size are normally fetched in the static or memory order. This will simplify the aligning and collapsing logic in the I-fetch unit.

Instructions are fetched by the I-fetch unit using a branch predictor capable of predicting multiple conditional branches simultaneously. Examples of this sort of predictor can be found in the work by Preston et al /cite-Seznec99,Preston02 The following heuristics are used for guding i-fetch :

If a conditional backward branch is predicted taken, the I-fetch unit will follow the taken path and continue loading instructions into the execution window for the mainline path from the target of the branch. A disjoint path

is also spawned to speculatively execute the instructions after the branch. This heuristic allows for the capture of program loops within the execution window of the machine.

In the case of forward branches, we de£ne two threshold values, *near threshold* and *far threshold*. We then compare the branch domain size with this threshold values which will categorize a branch as one of *near*, *midway* or *far*.

Since the domain of a forward near branch is small and can be easily loaded within the execution window, the fetch unit will load the instruction following the not-taken path of the branch, whether or not it is the predicted path. This represents fetching of instructions in the memory or static order rather than the program dynamic order and is very common in the absence of a loop. The fetching and loading of instructions following the not-taken output path of a conditional branch is very advantageous for capturing hammock styled branch constructs. Simple single-sided hammock branches are generally have near targets and are captured within the execution window. These branches are also good candidates for spawning a disjoint path which can further reduce the misprediction penalties.

If the branch is a midway branch, then based on its prediction con£dence, the instructions are fetched from either the target of the branch or subsequence instructions after the branch based on the outcome of the branch predictor.

In the case of a far branch, we will always follow the predicted path although if the branch is predicted not taken, there is an opportunity for spawning a disjoint path. The threshold values are parameters that can be either £xed at design time or adaptively changed during the course of execution.

D. Execution Control

In the last section we discussed as how I-fetch unit fetches instructions from I-cache along one or more predicted paths. Instructions are then decoded and staged into an instruction load buffer so that they are available to be loaded into the execution window. adjacent columns are loaded subsequently until the whole window is full. After that every time a column is committed and the its instructions are retired, a new set of instructions are loaded into that column. Branch mispredictions can have different effects on the execution. speculative mispredictions of branches that their domain is in the window results in a change of predicates and is handled through resource ¤ow execution. If the branch has a disjoint path and resolved to be mispredicted, a switch to the disjoint path is made. The last case is when the branch is resolved to be mispredicted and the instructions after the branch target had been fetched into the window. In this case the execution window has to be ¤*ushed* and fetch will resume after the branch.

A column can be committed when all the instructions in the column have been executed and there are no more transactions on any buses interconnecting the sharing groups in the column. As we mentioned before, there is no need for a centralized register £le in this architecture. The L1-D cache, however, needs to be updated with the store values not yet written back to it. The previous column buffers (PCB) in £gure ref£g:highlevel are used for this purpose. The PCB snoops the bus looking for the stores from the previous column and holds only the latest value for a given memory store address. PCB is used to distribute the store buffering throughout the machine and reduce the amount of buffered information.

E. Memory System

Throughout the paper, we introduced some of the features of this microprocessor that affect the memory system, including Memory Filter Units, L0 cache, PCB and memory operations in the context of resource-¤ow execution. The above features are mainly effective inside the execution window. A set of replicated copies of multi-ported non-blocking L1 D-cache are used to provide us with the necessary bandwidth and low miss rate. We also used a uni£ed L2 cache

IV. Evaluation

Baseline, IPC v.s. Geometry, Sensitivity data (Forwarding Units, Bus Span, Memory and L1), Effect of DEE, effect of MFU and L0 cache, execution window statistics, effect of changing the threshold values on the I-fetch performance.

Many machine sizes have been explored so far but only a subset of these sizes is further investigated in this paper. A particular machine is generally characterized using the tuple:

- sharing group rows
- active station rows per sharing group
- sharing group columns

These three characteristic parameters of a given machine are greatly in¤uential to its performance, as expected, and is termed the geometry of the machine. These three numbers are usually concatenated. For example the geometry of the machine in Figure 6 would be abbreviated 3-4-2.

REFERENCES

[1] Bannon P. et al. Internal architecture of alpha 21164 microprocessor. In *COMPCON'95*, pages 79–87, Mar 1995.

[2] Cher C. and Vijaykumar T.N. Skipper: A microarchitecture for exploiting control-ow independence. In *Proceedings of the 34nd International Symposium on Microarchitecture*, New York, NY, Nov 2001. ACM Press.

[3] Cleary J.G, Pearson M.W and Kinawi H. The architecture of an optimistic cpu: The warp engine. In *Proc. of the HICSS*, pages 163–172, Jan. 1995.

[4] Franklin M. and Sohi G.S. Arb: A hardware mechanism for dynamic memory disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[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] Gonzlez J. and Gonzlez A. Limits of instruction level parallelism with data speculation. In *Proceedings of the VECPAR Conference*, pages 585–598, 1998.

[7] Heil T.H. and Smith J.E. Selective dual path execution. Technical report, University of Wisconsin - Madison, Madison, WI, 1996.

[8] Kessler R.E., McLellan E.J. and Webb D.A. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, Oct 1998.

[9] 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*, pages 250–259, New York, NY, Jun 1998. ACM Press.

[10] Lam M.S. and Wilson R.P. Limits of control ow on parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, pages 46–57. ACM, May 1992.

[11] Lipasti M.H. and Shen J.P. Exceeding the data ow limit via value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, 1996.

[12] 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.

[13] Lipasti M.H and Shen J.P. Superspeculative microarchitecture for beyond AD 2000. *IEEE Computer*, 30(9):59–66, Sep 1997.

[14] Lipasti M.H and Shen J.P. Superspeculative microarchitecture for beyond ad 2000. *IEEE Computer*, 30(9), Sep 1997.

[15] 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.

[16] 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, 1997.

[17] 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, Jul 2001.

[18] Sankaranarayanan K. and Skadron K. A scheme for selective squash and re-issue for single-sided branch hammocks. IEEE Press, Oct 2001.

[19] Smith J.E. and Sohi G.S. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83:1609–1624, Dec 1995.

[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] Uht A.K. A theory of reduced and minimal procedural dependencies. *IEEE Transactions on Computers*, 40(6):681–692, Jun 1991.

[22] Uht A.K. and Sindagi V. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 313–325. ACM, Nov 1995.

[23] Uht A.K., Morano D.A., Khala£ 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, Dept. of ECE, URI, Sept 2001.

[24] Wallace S., Calder B. and Tullsen D.M. Threaded multiple path execution. In *Proceedings of the 25th International Symposium on Computer Architecture*, New York, NY, Jun 1998. ACM Press.

[25] 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.