Complexity-Effective Superscalar Processors

Subbarao Palacharla
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706, USA
subbarao@cs.wisc.edu

Norman P. Jouppi
Western Research Laboratory
Digital Equipment Corporation
Palo Alto, CA 94301, USA
jouppi@pa.dec.com

J. E. Smith
Dept. of Electrical and Computer Engg.
University of Wisconsin-Madison
Madison, WI 53706, USA
jes@ece.wisc.edu

Abstract

The performance tradeoff between hardware complexity and clock speed is studied. First, a generic superscalar pipeline is defined. Then the specific areas of register renaming, instruction window wakeup and selection logic, and operand bypassing are analyzed. Each is modeled and Spice simulated for feature sizes of $0.8\mu m$, $0.35\mu m$, and $0.18\mu m$. Performance results and trends are expressed in terms of issue width and window size. Our analysis indicates that window wakeup and selection logic as well as operand bypass logic are likely to be the most critical in the future.

A microarchitecture that simplifies wakeup and selection logic is proposed and discussed. This implementation puts chains of dependent instructions into queues, and issues instructions from multiple queues in parallel. Simulation shows little slowdown as compared with a completely flexible issue window when performance is measured in clock cycles. Furthermore, because only instructions at queue heads need to be awakened and selected, issue logic is simplified and the clock cycle is faster – consequently overall performance is improved. By grouping dependent instructions together, the proposed microarchitecture will help minimize performance degradation due to slow bypasses in future wide-issue machines.

1 Introduction

For many years, a major point of contention among microprocessor designers has revolved around complex implementations that attempt to maximize the number of instructions issued per clock cycle, and much simpler implementations that have a very fast clock cycle. These two camps are often referred to as "brainiacs" and "speed demons" – taken from an editorial in Microprocessor Report [7]. Of course the tradeoff is not a simple one, and through innovation and good engineering, it may be possible to achieve most, if not all, of the benefits of complex issue schemes, while still allowing a very fast clock in the implementation; that is, to develop microarchitectures we refer to as *complexity-effective*. One of two primary objectives of this paper is to propose such a complexity-effective microarchitecture. The proposed microarchitecture achieves high performance, as measured by instructions per cycle (IPC), yet it permits a design with a very high clock frequency.

Supporting the claim of high IPC with a fast clock leads to the second primary objective of this paper. It is commonplace to measure the effectiveness (i.e. IPC) of a new microarchitecture, typically by using trace driven simulation. Such simulations count clock cycles and can provide IPC in a fairly straightforward manner. However, the complexity (or simplicity) of a microarchitecture is much more difficult to determine – to be very accurate, it requires a full implementation in a specific technology. What is very much needed are fairly straightforward measures of complexity that can be used by microarchitects at a fairly early stage of the design process. Such methods would allow the determination of complexity-effectiveness. It is the second objective of this paper to take a step in the direction of characterizing complexity and complexity trends.

Before proceeding, it must be emphasized that while complexity can be variously quantified in terms such as number of transistors, die area, and power dissipated, in this paper complexity is measured as the delay of the critical path through a piece of logic, and the longest critical path through any of the pipeline stages determines the clock cycle.

The two primary objectives given above are covered in reverse order – first sources of pipeline complexity are analyzed, then a new complexity-effective microarchitecture is proposed and evaluated. In the next section we describe those portions of a microarchitecture that tend to have complexity that grows with increasing instruction-level parallelism. Of these, we focus on instruction dispatch and issue logic, and data bypass logic. We analyze potential critical paths in these structures and develop models for quantifying their delays. We study the variation of these delays with microarchitectural parameters of window size (the number of waiting instructions from which ready instructions are selected for issue) and the issue width (the number of instructions that can be issued in a cycle). We also study the impact of the technology trend towards smaller feature sizes. The complexity analysis shows that logic associated with the issue window and data bypasses are likely to be key limiters of clock speed since smaller feature sizes cause wire delays to dominate overall delay [20, 3].

Taking sources of complexity into account, we propose and evaluate a new microarchitecture. This microarchitecture is called *dependence-based* because it focuses on grouping dependent instructions rather than independent ones, as is often the case in superscalar implementations. The dependence-based microarchitecture simplifies issue window logic while exploiting similar levels of parallelism to that achieved by current superscalar microarchitectures using more complex logic.

The rest of the paper is organized as follows. Section 2 describes the sources of complexity in a baseline microarchitecture. Section 3 describes the methodology we use to study the critical pipeline structures identified in Section 2. Section 4 presents a detailed analysis of each of the structures and shows how their delays vary with microarchitectural parameters and technology parameters. Section 5 presents the proposed dependence-based microarchitecture and some preliminary performance results. Finally, we draw conclusions in Section 6.

2 Sources of Complexity

In this section, specific sources of pipeline complexity are considered. We realize that it is impossible to capture all possible microarchitectures in a single model, however, and any results have some obvious limitations. We can only hope to provide a fairly straightforward model that is typical of most current superscalar processors, and suggest that analyses similar to those used here can be extended to other, more advanced techniques as they are developed.

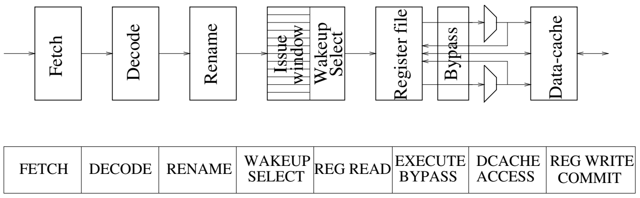

Figure 1: Baseline superscalar model.

Figure 1 shows the baseline model and the associated pipeline. The fetch unit reads multiple instructions every cycle from the instruction cache, and branches encountered by the fetch unit are predicted. Next, instructions are decoded and their register operands are renamed. Renamed instructions are dispatched to the instruction window, where they wait for their source operands and the appropriate functional unit to become available. As soon as these conditions are satisfied, instructions are issued and executed in the functional units. The operand values of an instruction are either fetched from the register file or are bypassed from earlier instructions in the pipeline. The data cache provides low latency access to memory operands.

2.1 Basic Structures

As mentioned earlier, probably the best way to identify the primary sources of complexity in a microarchitecture is to actually implement the microarchitecture in a specific technology. However, this is extremely time consuming and costly. Instead, our approach is to select certain key structures for study, and develop relatively simple delay models that can be applied in a straightforward manner without relying on detailed design.

Structures to be studied were selected using the following criteria. First, we consider structures whose delay is a function of issue window size and/or issue width; these structures are likely to become cycle-time limiters in future wide-issue superscalar designs. Second, we are interested in dispatch and issue-related structures because these structures form the core of a microarchitecture and largely determine the amount of parallelism that can be exploited. Third, some structures tend to rely on broadcast operations over long wires and hence their delays might not scale as well as logic-intensive structures in future technologies with smaller feature sizes.

The structures we consider are:

- *Register rename logic*. This logic translates logical register designators into physical register designators.
- *Wakeup logic*. This logic is part of the issue window and is responsible for waking up instructions waiting for their source operands to become available.
- *Selection logic*. This logic is another part of the issue window and is responsible for selecting instructions for execution from the pool of ready instructions.
- *Bypass logic*. This logic is responsible for bypassing operand values from instructions that have completed execution, but have not yet written their results to the register file, to subsequent instructions.

There are other important pieces of pipeline logic that are not considered in this paper, even though their delay is a function of dispatch/issue width. In most cases, their delay has been considered elsewhere. These include register files and caches. Farkas et. al. [6] study how the access time of the register file varies with the number of registers and the number of ports. The access time of a cache is a function of the size of the cache and the associativity of the cache. Wada et. al. [18] and Wilton and Jouppi [21] have developed detailed models that estimate the access time of a cache given its size and associativity.

2.2 Current Implementations

The structures identified above were presented in the context of the baseline superscalar model shown in Figure 1. The MIPS R10000 [22] and the DEC 21264 [10] are real implementations that directly fit this model. Hence, the structures identified above apply to these two processors.

On the other hand, the Intel Pentium Pro [9], the HP PA-8000 [12], the PowerPC 604 [16], and the HAL SPARC64 [8] do not completely fit the baseline model. These processors are based on a microarchitecture where the reorder buffer holds non-committed, renamed register values. In contrast, the baseline microarchitecture uses the physical register file for both committed and non-committed values. Nevertheless, the point to be noted is that the basic structures identified earlier are present in both types of microarchitectures. The only notable difference is the size of the physical register file.

Finally, while the discussion about potential sources of complexity is in the context of an out-of-order baseline superscalar model, it must be pointed out that some of the critical structures identified apply to in-order processors, too. For example, part of the register rename logic (to be discussed later) and the bypass logic are present in in-order superscalar processors.

3 Methodology

The key pipeline structures were studied in two phases. In the first phase, we selected a representative CMOS circuit for the structure. This was done by studying designs published in the literature (e.g. ISSCC [1] proceedings) and by collaborating with engineers at Digital Equipment Corporation. In cases where there was more than one possible design, we did a preliminary study of the designs to decide in favor of one that was most promising. By basing our circuits on designs published by microprocessor vendors, we believe the studied circuits are similar to circuits used in microprocessor designs. In practice, many circuit tricks could be employed to optimize critical paths. However, we believe that the relative delays between different structures should be more accurate than the absolute delays.

[1] International Solid-State and Circuits Conference.

In the second phase we implemented the circuit and optimized the circuit for speed. We used the Hspice circuit simulator [14] from Meta-Software to simulate the circuits. Primarily, static logic was used. However, in situations where dynamic logic helped in boosting the performance significantly, we used dynamic logic. For example, in the wakeup logic, a dynamic 7-input NOR gate is used for comparisons instead of a static gate. A number of optimizations were applied to improve the speed of the circuits. First, all the transistors in the circuit were manually sized so that overall delay improved. Second, logic optimizations like two-level decomposition were applied to reduce fan-in requirements. We avoided using static gates with a fan-in greater than four. Third, in some cases transistor ordering was modified to shorten the critical path. Wire parasitics were added at appropriate nodes in the Hspice model of the circuit. These parasitics were computed by calculating the length of the wires based on the layout of the circuit and using the values of R_{metal} and C_{metal}, the resistance and parasitic capacitance of metal wires per unit length.

To study the effect of reducing the feature size on the delays of the structures, we simulated the circuits for three different feature sizes: $0.8\mu m$, $0.35\mu m$, and $0.18\mu m$ respectively. Layouts for the $0.35\mu m$ and $0.18\mu m$ process were obtained by appropriately shrinking the layouts for the $0.8\mu m$ process. The Hspice models used for the three technologies are tabulated in [15].

4 Pipeline Complexity

In this section, we analyze the critical pipeline structures. The presentation for each structure begins with a description of the logical function being implemented. Then, possible implementation schemes are discussed, and one is chosen. Next, we summarize our analysis of the overall delay in terms of the microarchitectural parameters of issue width and issue window size; a much more detailed version of the analysis appears in [15]. Finally, Hspice circuit simulation results are presented and trends are identified and compared with the earlier analysis.

4.1 Register Rename Logic

Register rename logic translates logical register designators into physical register designators by accessing a map table with the logical register designator as the index. The map table holds the current logical to physical mappings and is multi-ported because multiple instructions, each with multiple register operands, need to be renamed every cycle. The high level block diagram of the rename logic is shown in Figure 2. In addition to the map table, dependence check logic is required to detect cases where the logical register being renamed is written by an earlier instruction in the current group of instructions being renamed. The dependence check logic detects such dependences and sets up the output MUXes so that the appropriate physical register designators are selected. At the end of every rename operation, the map table is updated to reflect the new logical to physical mappings created for the result registers written by the current rename group.

4.1.1 Structure

The mapping and checkpointing functions of the rename logic can be implemented in at least two ways. These two schemes, called the RAM scheme and the CAM scheme, are described next.

- *RAM scheme*
  In the RAM scheme, implemented in the MIPS R10000 [22], the map table is a register file where the logical register designator directly accesses an entry that contains the physical register to which it is mapped. The number of entries in the map table is equal to the number of logical registers.

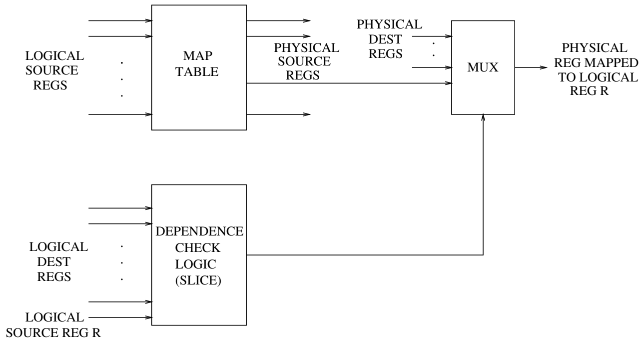

Figure 2: Register rename logic.

- *CAM scheme*
  An alternate scheme for register renaming uses a CAM (content-addressable memory) [19] to store the current mappings. Such a scheme is implemented in the HAL SPARC [2] and the DEC 21264 [10]. The number of entries in the CAM is equal to the number of physical registers. Each entry contains two fields: the logical register designator that is mapped to the physical register represented by the entry and a valid bit that is set if the current mapping is valid. Renaming is accomplished by matching on the logical register designator field.

In general, the CAM scheme is less scalable than the RAM scheme because the number of CAM entries, which is equal to the number of physical registers, tends to increase with issue width. Also, for the design space we are interested in, the performance was found to be comparable. Consequently, we focus on the RAM method below. A more detailed discussion of the trade-offs involved can be found in [15].

The dependence check logic proceeds in parallel with the map table access. Every logical register designator being renamed is compared against the logical destination register designators of earlier instructions in the current rename group. If there is a match, then the physical register assigned to the result of the earlier instruction is used instead of the one read from the map table. In the case of multiple matches, the register corresponding to the latest (in dynamic order) match is used. Dependence check logic for issue widths of 2, 4, and 8 was implemented. We found that for these issue widths, the delay of the dependence check logic is less than the delay of the map table, and hence the check can be hidden behind the map table access.

4.1.2 Delay Analysis

As the name suggests, the RAM scheme operates like a standard RAM. Address decoders drive word lines; an access stack at the addressed cell pulls a bitline low. The bitline changes are sensed by a sense amplifier which in turn produces the output. Symbolically the rename delay can be written as,

$$T_{rename} = T_{decode} + T_{wordline} + T_{bitline} + T_{senseamp}$$

The analysis presented here and in following subsections focuses on those parts of the delay that are a function of the issue width and window size. All sources of delay are considered in detail in [15]. In the rename logic, the window size is not a factor, and the issue width affects delay through its impact on wire lengths. Increasing the issue width increases the number of bitlines and wordlines in each cell thus making each cell bigger. This in turn increases the length of the predecode, wordline, and bitline wires and the associated wire delays. The net effect is the following relationships for the delay components:

$$T_{decode}, T_{wordline}, T_{bitline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

where IW is the issue width and c_0, c_1, and c_2 are constants that are fixed for a given technology and instruction set architecture; derivation of the constants for each component is given in [15]. In each case, the quadratic component, resulting from the intrinsic RC delay of wires, is relatively small for the design space and technologies we explored. Hence, the decode, wordline, and bitline delays are effectively linear functions of the issue width.

For the sense amplifier, we found that even though its structural constitution is independent of the issue width, its delay is a function of the slope of the input – the bitline delay – and therefore varies linearly with issue width.

4.1.3 Spice Results

For our Hspice simulations, Figure 3 shows how the delay of the rename logic varies with the issue width i.e. the number of instructions being renamed every cycle for the three technologies. The graph includes the breakdown of delay into components discussed in the previous section.

A number of observations can be made from the graph. The total delay increases linearly with issue width for all the technologies. This is in conformance with our analysis, summarized in the previous section. Furthermore, each of the components shows a linear increase with issue width. The increase in the bitline delay is larger than the increase in the wordline delay as issue width is increased because the bitlines are longer than the wordlines in our design. The bitline length is proportional to the number of logical registers (32 in most cases) whereas the wordline length is proportional to the width of the physical register designator (less than 8 for the design space we explored).

Another important observation that can be made from the graph is that the relative increase in wordline delay, bitline delay, and hence, total delay as a function of issue width worsens as the feature size is reduced. For example, as the issue width is increased from 2 to 8, the percentage increase in bitline delay shoots up from 37% to 53% as the feature size is reduced from $0.8\mu m$ to $0.18\mu m$. Logic delays in the various components are reduced in proportion to the feature size, while the presence of wire delays in the wordline and bitline components cause the wordline and bitline components to fall at a slower rate. In other words, wire delays in the wordline and bitline structures will become increasingly important as feature sizes are reduced.

4.2 Wakeup Logic

Wakeup logic is responsible for updating source dependences for instructions in the issue window waiting for their source operands to become available.

4.2.1 Structure

Wakeup logic is illustrated in Figure 4. Every time a result is produced, a tag associated with the result is broadcast to all the instructions in the issue window. Each instruction then compares the tag with the tags of its source operands. If there is a match, the operand is marked as available by setting the rdyL or rdyR flag. Once all the operands of an instruction become available (both rdyL and rdyR are set), the instruction is ready to execute, and the ready flag is set

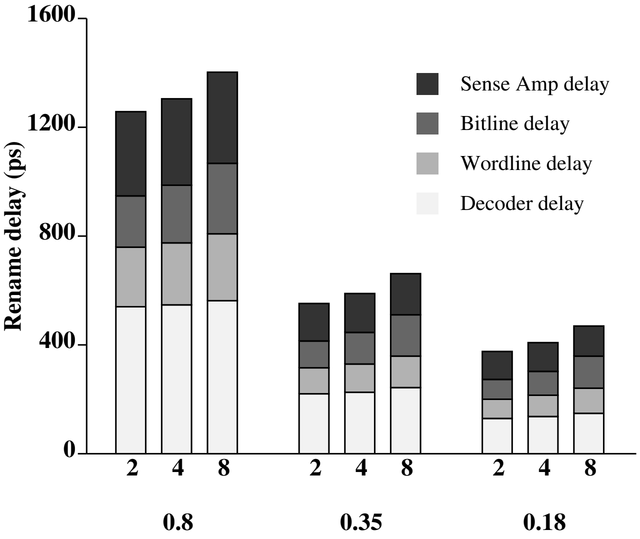

Figure 3: Rename delay versus issue width.

to indicate this. The issue window is a CAM array holding one instruction per entry. Buffers, shown at the top of the figure, are used to drive the result tags $tag1$ to $tagIW$, where IW is the issue width. Each entry of the CAM has $2 \times IW$ comparators to compare each of the results tags against the two operand tags of the entry. The OR logic ORs the comparator outputs and sets the rdyL/rdyR flags.

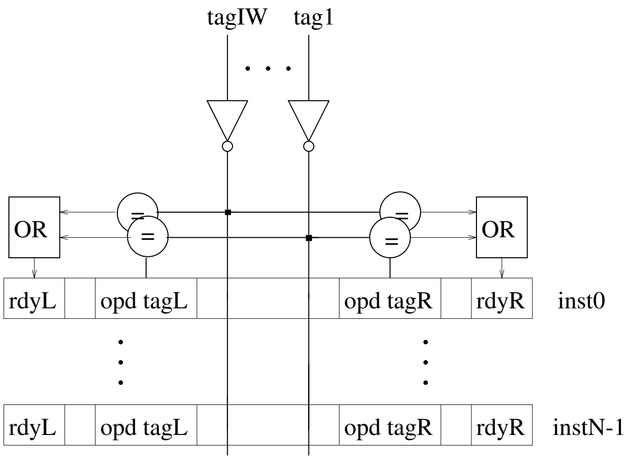

Figure 4: Wakeup logic.

4.2.2 Delay Analysis

The delay consists of three components: the time taken by the buffers to drive the tag bits, the time taken by the comparators in a pull-down stack corresponding to a mismatching bit position to pull the matchline low [2], and the time taken to OR the individual match signals (matchlines). Symbolically,

$$Delay = T_{tagdrive} + T_{tagmatch} + T_{matchOR}$$

The time taken to drive the tags depends on the length of the tag lines and the number of comparators on the tag lines. Increasing the window size increases both these terms. For a given window size,

[2] We assume that only one pull-down stack is turned on since we are interested in the worst-case delay.

increasing issue width also increases both the terms in the following way. Increasing issue width increases the number of matchlines in each cell and hence increases the height of each cell. Also, increasing issue width increases the number of comparators in each cell. Note that we assume the maximum number of tags produced per cycle is equal to the maximum issue width.

In simplified form (see [15] for a more detailed analysis), the time taken to drive the tags is:

$$T_{tagdrive} = c_0 + (c_1 + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c_5 \times IW^2) \times WINSIZE^2$$

The above equation shows that the tag drive time is a quadratic function of the window size. The weighting factor of the quadratic term is a function of the issue width. The weighting factor becomes significant for issue widths beyond 2. For a given window size, the tag drive time is also a quadratic function of the issue width. For current technologies ($0.35\mu m$ and longer) the quadratic component is relatively small and the tag drive time is largely a linear function of issue width. However, as the feature size is reduced to $0.18\mu m$, the quadratic component also increases in significance. The quadratic component results from the intrinsic RC delay of the tag lines.

In reality, both issue width and window size will be simultaneously increased because a larger window is required for finding more independent instructions to take advantage of wider issue. Hence, the tag drive time will become significant in future designs with wider issue widths, bigger windows, and smaller feature sizes.

The tag match time is primarily a function of the length of the matchline, which varies linearly with the issue width. The match OR time is the time taken to OR the match lines, and the number of matchlines is a linear function of issue width. Both of these (refer to [15]) have a delay:

$$T_{tagmatch}, T_{matchOR} = c_0 + c_1 \times IW + c_2 \times IW^2$$

However, in both cases the quadratic term is very small for the design space we consider, so these delays are linear functions of issue width.

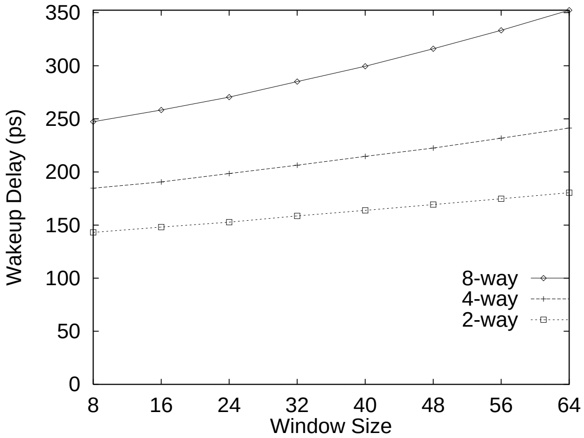

Figure 5: Wakeup logic delay versus window size.

4.2.3 Spice Results

The graph in Figure 5 shows how the delay of the wakeup logic varies with window size and issue width for $0.18\mu m$ technology. As expected, the delay increases as window size and issue width are increased. The quadratic dependence of the total delay on the window size results from the quadratic increase in tag drive time as discussed in the previous section. This effect is clearly visible for issue width of 8 and is less significant for issue width of 4. We found similar curves for $0.8\mu m$ and $0.35\mu m$ technologies. The quadratic dependence of delay on window size was more prominent in the curves for $0.18\mu m$ technology than in the case of the other two technologies.

Also, issue width has a greater impact on the delay than window size because increasing issue width increases all three components of the delay. On the other hand, increasing window size only lengthens the tag drive time and to a small extent the tag match time. Overall, the results show that the delay increases by almost 34% going from 2-way to 4-way and by 46% going from 4-way to 8-way for a window size of 64 instructions. In reality, the increase in delay is going to be even worse because in order to sustain a wider issue width, a larger window is required to find independent instructions.

Figure 6 shows the effect of reducing feature sizes on the various components of the wakeup delay for an 8-way, 64-entry window processor. The tag drive and tag match delays do not scale as well as the match OR delay. This is expected since tag drive and tag match delays include wire delays whereas the match OR delay only consists of logic delays. Quantitatively, the fraction of the total delay contributed by tag drive and tag match delay increases from 52% to 65% as the feature size is reduced from $0.8\mu m$ to $0.18\mu m$. This shows that the performance of the broadcast operation will become more crucial in future technologies.

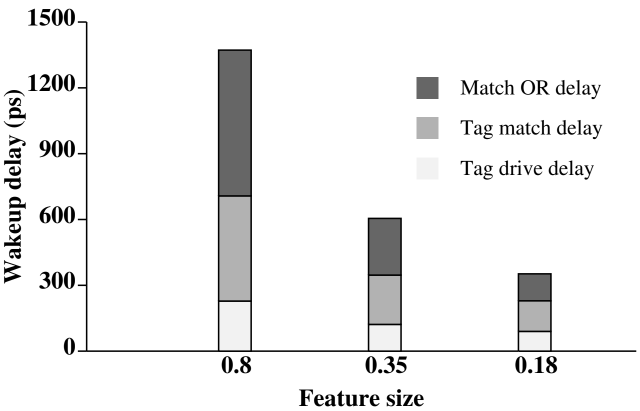

Figure 6: Wakeup delay versus feature size.

4.3 Selection Logic

Selection logic is responsible for choosing instructions for execution from the pool of ready instructions in the issue window. Some form of selection logic is required because the number and types of ready instructions may exceed the number and types of functional units available to execute them.

Inputs to the selection logic are request (REQ) signals, one per instruction in the issue window. The request signal of an instruction is raised when the wakeup logic determines that all its operands are available. The outputs of the selection logic are grant (GRANT) signals, one per request signal. On receipt of the GRANT signal, the associated instruction is issued to the functional unit.

A *selection policy* is used to decide which of the requesting instructions is granted. An example selection policy is *oldest first* - the ready instruction that occurs earliest in program order is granted the functional unit. Butler and Patt [5] studied various policies for scheduling ready instructions and found that overall performance is largely independent of the selection policy. The HP PA-8000 uses a selection policy that is based on the location of the instruction in the window. We assume the same selection policy in our study.

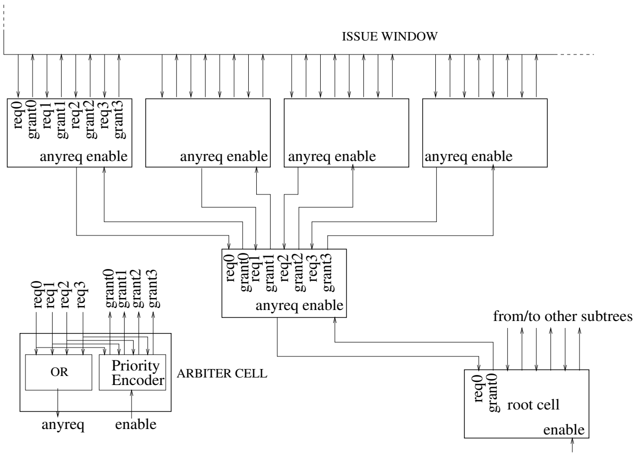

Figure 7: Selection logic.

4.3.1 Structure

The basic structure of selection logic is shown in Figure 7. Modifications to this scheme for handling multiple functional units of the same type are discussed in [15]. Selection logic consists of a tree of arbiters that works in two phases. In the first phase, request signals are propagated up the tree. Each cell raises the $anyreq$ signal if any of its input request signals is high. This in turn raises the input request signal of its parent arbiter cell. At the root cell one or more of the input request signals will be high if there are one or more instructions that are ready. The root cell grants the functional unit to one of its children by raising one of its grant outputs. This initiates the second phase where the grant signal is propagated down the tree to the instruction that is selected. The enable signal to the root cell is high whenever the functional unit is ready to execute an instruction.

The selection policy implemented is static and based strictly on location of the instruction in the issue window. The leftmost entries in the window have the highest priority. The *oldest first* policy can be implemented using this scheme by compacting the issue window to the left every time instructions are issued and by inserting new instructions at the right end. However, it is possible that the complexity of compaction could degrade performance. In this case, some restricted form of compacting can be used – so that overall performance is not affected. We did not analyze the complexity of compacting in this study.

4.3.2 Delay Analysis

The delay of the selection logic is the time it takes to generate the grant signal after the request signal has been raised. This is equal to the sum of three terms: the time taken for the request signal to propagate to the root of the tree, the delay of the root cell, and the time taken for the grant signal to propagate from the root to the selected instruction. Hence, the selection delay depends on the height of the arbitration tree and can be written as (see [15] for a more detailed analysis):

$$T_{selection} = c_0 + c_1 \times log_4(WINSIZE)$$

where c_0 and c_1 are constants determined by the propagation delays of a single arbiter. We found the optimal number of arbiter inputs to be four in our case, so the logarithm is base 4. The selection logic in the MIPS R10000, described in [17], is also based on four-input arbiter cells.

4.3.3 Spice Results

Figure 8 shows the delay of the selection logic for various window sizes and for the three feature sizes assuming a single functional unit is being scheduled. The delay is broken down into the three components. From the graph we can see that for all the three technologies, the delay increases logarithmically with window size. Also, the increase in delay is less than 100% when the window size is increased from 16 instructions to 32 instructions (or from 64 instructions to 128 instructions) since one of the components of the total delay, the delay at the root cell, is independent of the window size.

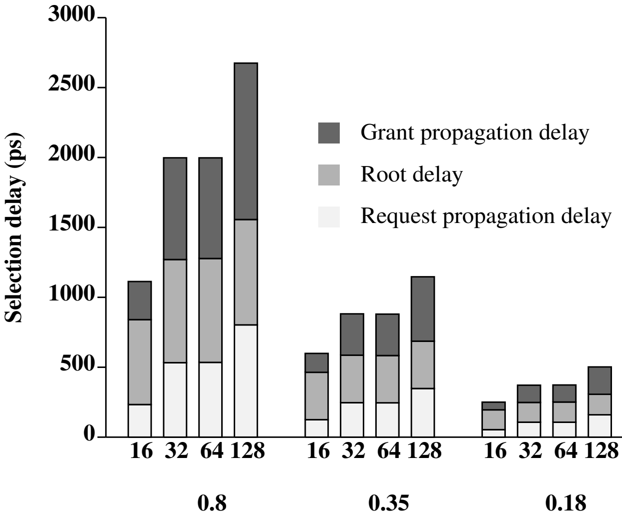

Figure 8: Selection delay versus window size.

The various components of the total delay scale well as the feature size is reduced. This is not surprising since all the delays are logic delays. It must be pointed out that we do not consider the wires in the circuit, so the selection delays presented here are optimistic, especially if the request signals (the ready flags discussed in the wakeup logic) originate from the CAM entries in which the instructions reside. On the other hand, it might be possible to minimize the effect of these wire delays if the ready signals are stored in a smaller, more compact array.

4.4 Data Bypass Logic

Data bypass logic is responsible for forwarding result values from completing instructions to dependent instructions, bypassing the register file. The number of bypass paths required is determined by the depth of the pipeline and the issue width of the microarchitecture. As pointed out in [1], if IW is the issue width, and if there are S pipestages after the first result-producing stage, then a fully bypassed design would require $(2 \times IW^2 \times S)$ bypass paths assuming 2-input functional units. In other words, the number of bypass paths grows quadratically with issue width. This is of critical importance, given the current trends toward deeper pipelines and wider issue.

Bypass logic consists of two components: datapath and control. The datapath comprises result busses, that are used to broadcast bypass values from each functional unit source to all possible destinations. Buffers are used to drive the bypass values on the result busses. In addition to the result busses, the datapath comprises operand MUXes. Operand MUXes are required to gate in the appropriate result on to the operand busses. The control logic is responsible for controlling the operand MUXes. It compares the tags of the result values with the tag of source value required at each functional unit. If there is a match, the MUX control is set so that the result value is driven on the appropriate operand bus. The key factor that determines the speed of the bypass logic is the delay of the result wires that are used to transmit bypassed values, not the control.

4.4.1 Structure

A commonly used structure for bypass logic is shown in Figure 9. The figure shows a bit-slice of the datapath. There are four functional units marked FU0 to FU3. Consider the bit slice of FU0. It gets its two operand bits from the *opd0-l* and *opd0-r* wires. The result bit is driven on the *res0* result wire by the result driver. Tristate buffers are used to drive the result bits on to the operand wires from the result wires of the functional units. These buffers implement the MUXes shown in the figure. To bypass the result of functional unit FU1 to the left input of functional unit FU0, the tristate driver marked A is switched on. The driver A connects the *res1* wire and the *opd0-l* wire. In the case where bypasses are not activated, operand bits are placed on the operand wires by the register file read ports [3]. The result bits are written to the register file in addition to being bypassed.

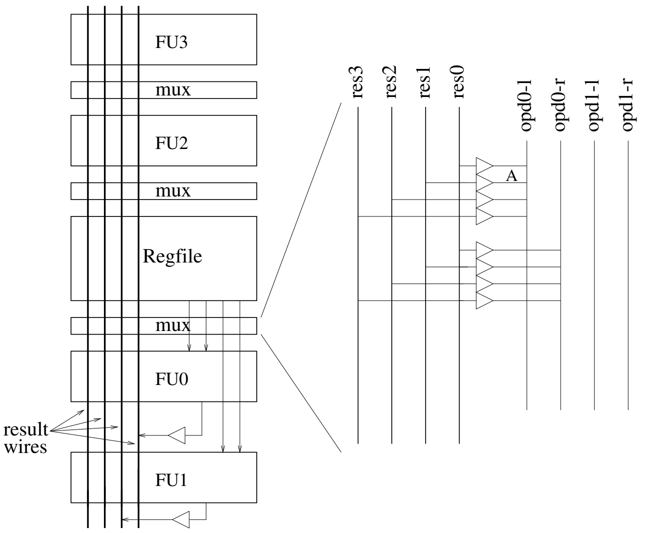

Figure 9: Bypass logic.

4.4.2 Delay Analysis

The delay of the bypass logic is largely determined by the amount of time it takes for the driver at the output of each functional unit to drive the result value on the corresponding result wire. This in turn depends on the length of the result wires. From the figure it is apparent that the length of the wires is a function of the layout. For the layout presented in the figure, the length of the result wires is determined by the height of the functional units and the register file.

[3] In a reservation-station based microarchitecture, the operand bits come from the data field of the reservation station entry.

Issue width	Wire length (λ)	Delay (ps)
4	20500	184.9
8	49000	1056.4

Table 1: Bypass delays for a 4-way and a 8-way processor.

Considering the result wires as distributed RC lines, the delay is given by

$$T_{bypass} = 0.5 \times R_{metal} \times C_{metal} \times L^2$$

where L is the length of the result wires, and R_{metal} and C_{metal} are the resistance and parasitic capacitance of metal wires per unit length respectively.

Increasing issue width increases the length of the result wires, and hence causes the bypass delay to grow quadratically with issue width. Increasing the depth of the pipeline also increases the delay of the bypass logic in the following manner. Making the pipeline deeper increases the fan-in of the operand MUXes connected to a given result wire. This in turn increases the amount of capacitance on the result wires, and hence adds to the delay of the result wires. However, this component of the delay is not captured by our simple model. This component of the delay is likely to become relatively less significant as feature size is reduced.

4.4.3 Spice Results

We computed the wire delays for hypothetical 4-way and 8-way machines assuming common mixes of functional units and functional unit heights reported in the literature. Table 1 shows the results. Wire lengths are shown in terms of λ, where λ is half the feature size. The delays are the same for the three technologies since wire delays are constant according to the scaling model assumed. See [15] for the detailed data and analysis.

4.4.4 Alternative Layouts

The results presented above assume a particular layout; the functional units are placed on either side of the register file. However, as mentioned before, the length of the result wires is a function of the layout. Hence, VLSI designers will have to study alternative layouts in order to reduce bypass delays. Alternative layouts alone will only decrease constants; the quadratic delay growth with number of bypasses will remain.

In the long term, microarchitects will have to consider *clustered* organizations where each cluster of functional units has its own copy of the register file and bypasses within a cluster complete in a single cycle while inter-cluster bypasses take two or more cycles. The hardware or the compiler or both will have to ensure that inter-cluster bypasses occur infrequently. In addition to mitigating the delay of the bypass logic, this organization also has the advantage of faster register files since there are fewer ports on each register file.

4.5 Summary of Delays and Pipeline Issues

We now summarize the pipeline delay results and consider the implications for future complexity-effective microarchitectures. It is easiest to frame this discussion in terms of satisfying the goal of permitting a very fast pipeline clock while, at the same time, exploiting high ILP through relatively wide, out-of-order superscalar operation.

Issue width	Window size	Rename delay (ps)	Wakeup+Select delay (ps)	Bypass delay (ps)
0.8μm technology				
4	32	1577.9	2903.7	184.9
8	64	1710.5	3369.4	1056.4
0.35μm technology				
4	32	627.2	1248.4	184.9
8	64	726.6	1484.8	1056.4
0.18μm technology				
4	32	351.0	578.0	184.9
8	64	427.9	724.0	1056.4

Table 2: Overall delay results.

To aid in this discussion, consider the overall results for a 4-way and a 8-way microarchitecture in $0.18\mu m$ technology shown in Table 2. We chose the $0.18\mu m$ technology because of our interest in future generation microarchitectures. For the 4-way machine, the window logic (wakeup + select) has the greatest delay among all the structures considered, and hence determines the critical path delay. The register rename delay comes next; it is about 39% faster than the delay of the window logic. The bypass delay is relatively small in this case. The results are similar for the 8-way machine, with one very notable exception: the bypass delay grows by a factor of over 5, and is now worse than the (wakeup + select) delay.

Now, let's turn to the problem of designing a future generation microarchitecture with a faster clock cycle. Of the structures we have examined here, the window logic and the bypasses seem to pose the largest problems. Moreover, both of these cause difficulties if we wish to divide them into more pipeline segments; these difficulties will be discussed in the following paragraphs. All the other structures either will not cause a clock cycle problem, or if they do, they can be pipelined. The pipelining aspects of these structures is discussed in [15]. This additional pipelining can cause some performance impact, although it is beyond the scope of this paper to evaluate the exact impact.

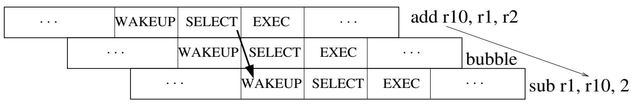

Figure 10: Pipelining wakeup and select.

Wakeup and select together constitute what appears to be an *atomic* operation. That is, if they are divided into multiple pipeline stages, dependent instructions cannot issue in consecutive cycles. Consider the pipeline example shown in Figure 10. The `add` and the `sub` instructions cannot be executed back-to-back because the result of the select stage has to feed the wakeup stage. Hence, wakeup and select together constitute an atomic operation and must be accomplished in a single cycle, at least if dependent instructions are to be executed on consecutive cycles.

Data bypassing is another example of what appears to be an atomic operation. In order for dependent operations to execute in consecutive cycles, the bypass value must be made available to the dependent instruction within a cycle. Results presented in table Table 2 show that this is feasible for a 4-way machine. However, bypass delay can easily become a bottleneck for wider issue-widths.

One solution is to include only a proper subset of bypass paths [1], and take a penalty for those that are not present. For an 8-way machine with deep pipelines, this would exclude a large number of bypass paths. Another solution is to generalize the method used in the DEC 21264 [10] and use multiple copies of the register file. This is the "cluster" method referred to in Section 4.4.

In the following section we tackle both the window logic and bypass problems by proposing a microarchitecture that simplifies window logic and which naturally supports clustering of functional units.

5 A Complexity-Effective Microarchitecture

From the analysis presented in the previous sections we see that the issue window logic is one of the primary contributors of complexity in typical out-of-order microarchitectures. Also, as architects employ wider issue-widths and deeper pipelines, the delay of the bypass logic becomes even more critical. In this section, we propose a *dependence-based* microarchitecture that replaces the issue window with a simpler structure that facilitates a faster clock while exploiting similar levels of parallelism. In addition, the proposed microarchitecture naturally lends itself to clustering and helps mitigate the bypass problem to a large extent.

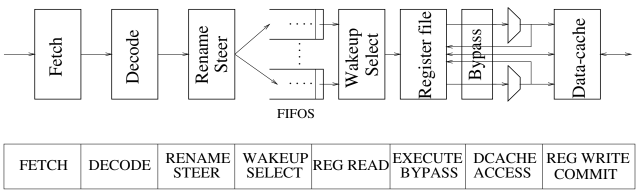

Figure 11: Dependence-based microarchitecture.

The idea behind the dependence-based microarchitecture is to exploit the natural dependences among instructions. A key point is that dependent instructions cannot execute in parallel. In the proposed microarchitecture, shown in Figure 11, the issue window is replaced by a small number of FIFO buffers. The FIFO buffers are constrained to issue in-order, and dependent instructions are steered to the same FIFO. This ensures that the instructions in a particular FIFO buffer can only execute sequentially. Hence, unlike the typical issue window where result tags have to be broadcast to all the entries, the register availability only needs to be fanned out to the heads of the FIFO buffers. The instructions at the FIFO heads monitor reservation bits (one per physical register) to check for operand availability. This is discussed in detail later. Furthermore, the selection logic only has to monitor instructions at the heads of the FIFO buffers.

The steering of dependent instructions to the FIFO buffers is performed at run-time during the rename stage. Dependence information between instructions is maintained in a table called the `SRC_FIFO` table. This table is indexed using logical register designators. `SRC_FIFO(Ra)`, the entry for logical register `Ra`, contains the identity of the FIFO buffer that contains the instruction that will write register `Ra`. If that instruction has already completed i.e. register `Ra` contains its computed value, then `SRC_FIFO(Ra)` is invalid. This table is similar to the map table used for register renaming and can be accessed in parallel with the rename table. In order to steer an instruction to a particular FIFO, the `SRC_FIFO` table is accessed with the register identifiers of the source operands of an instruction. For example, for steering the instruction `add r10,r5,1` where `r10` is the destination register, the SRC_FIFO table is indexed with 5. The entry is then used to steer the instruction to the appropriate FIFO.

5.1 Instruction Steering Heuristics

A number of heuristics are possible for steering instructions to the FIFOs. A simple heuristic that we found to work well for our benchmark programs is described next.

Let I be the instruction under consideration. Depending upon the availability of I's operands, the following cases are possible:

- All the operands of I have already been computed and are residing in the register file. In this case, I is steered to a new (empty) FIFO acquired from a pool of free FIFOs.
- I requires a single outstanding operand to be produced by instruction Isource residing in FIFO Fa. In this case, if there is no instruction behind Isource in Fa then I is steered to Fa, else I is steered to a new FIFO.
- I requires two outstanding operands to be produced by instructions Ileft and Iright residing in FIFOs Fa and Fb respectively. In this case, apply the heuristic in the previous bullet to the left operand. If the resulting FIFO is not suitable (it is either full or there is an instruction behind the source instruction), then apply the same heuristic to the right operand.

If all the FIFOs are full or if no empty FIFO is available then the decoder/steering logic stalls. A FIFO is returned to the free pool when the last instruction in the FIFO is issued. Initially, all the FIFOs are in the free pool. Figure 12 illustrates the heuristic on a code segment from one of the SPEC benchmarks.

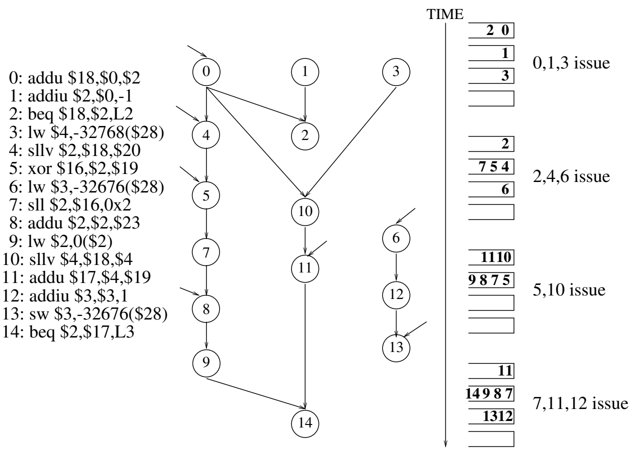

Figure 12: Instruction steering example.

This figure shows how instructions are steered to FIFOs using the heuristic presented in Section 5.1 for a sample code segment. Instructions can issue only from the heads of the four FIFOs. The steering logic steers four instructions every cycle and a maximum of four instructions can issue every cycle.

5.2 Performance Results

We compare the performance of the dependence-based microarchitecture against that of a typical microarchitecture with an issue window. The proposed microarchitecture has 8 FIFOs, with each FIFO having 8-entries. The issue window of the conventional processor has 64 entries. Both microarchitectures can decode, rename, and execute a maximum of 8 instructions per cycle. The timing simulator, a modified version of SimpleScalar [4], is detailed in Table 3.

Fetch width	any 8 instructions
I-cache	Perfect instruction cache
Branch Predictor	McFarling's gshare [13] 4K 2-bit counters, 12 bit history unconditional control instructions predicted perfectly
Issue window size	64
Max. in-flight instructions	128
Retire width	16
Functional Units	8 symmetrical units
Functional Unit Latency	1 cycle
Issue Mechanism	out-of-order issue of up to 8 ops/cycle loads may execute when all prior store addresses are known
Physical Registers	120 int/120 fp
D-cache	32KB, 2-way SA write-back, write-allocate 32 byte lines,1 cycle hit,6 cycle miss four load/store ports

Table 3: Baseline simulation model.

An aggressive instruction fetch mechanism is used to stress the issue and execution subsystems. We ran seven benchmarks from the SPEC'95 suite, using their training input datasets. Each benchmark was run for a maximum of 0.5B instructions.

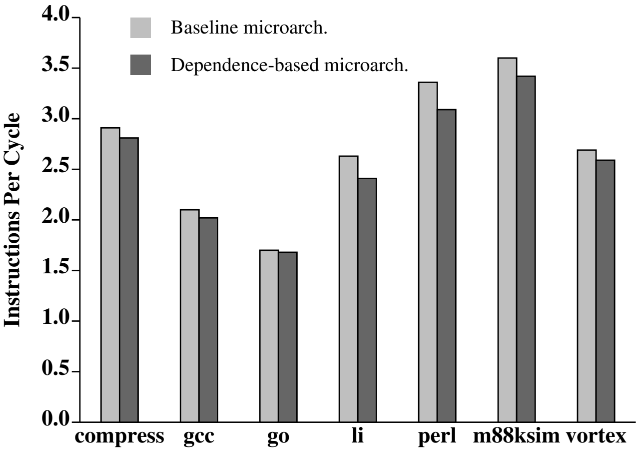

Figure 13: Performance (IPC) of dependence-based microarchitecture.

The performance results (in terms of instructions committed per cycle) are shown in Figure 13. The dependence-based microarchitecture is nearly as effective (extracts similar parallelism) as the typical window-based microarchitecture. The cycle count numbers are within 5% for five of the seven benchmarks and the maximum performance degradation is 8% in the case of *li*.

5.3 Complexity Analysis

First, consider the delay of the wakeup and selection logic. Wakeup logic is required to detect cross-FIFO dependences. For example, if the instruction Ia at the head of FIFO Fa is dependent on an instruction Ib waiting in FIFO Fb, then Ia cannot issue until Ib completes. However, the wakeup logic in this case does not involve broadcasting the result tags to all the waiting instructions. Instead, only the instructions at the FIFO heads have to determine when all

Issue width	No. physical registers	No. table entries	Bits per entry	Total delay (ps)
4	80	10	8	192.1
8	128	16	8	251.7

Table 4: Delay of reservation table in $0.18\mu m$ technology.

their operands are available. This is accomplished by interrogating a table called the *reservation table*. The reservation table contains a single bit per physical register that indicates whether the register is waiting for its data. When an instruction is dispatched, the reservation bit corresponding to its result register is set. The bit is cleared when the instruction executes and the result value is produced. An instruction at the FIFO head waits until the reservation bits corresponding to its operands are cleared. Hence, the delay of the wakeup logic is determined by the delay of accessing the reservation table. The reservation table is relatively small in size compared to the rename table and the register file. For example, for a 4-way machine with 80 physical registers, the reservation table can be laid out as a 10-entry table with each entry storing 8 bits [4]. Table 4 shows the delay of the reservation table for 4-way and 8-way machines. For both cases, the wakeup delay is much smaller than the wakeup delay for a 4-way, 32-entry issue window-based microarchitecture. Also, this delay is smaller than the corresponding register renaming delay. The selection logic in the proposed microarchitecture is simple because only the instructions at the FIFO heads need to be considered for selection.

Instruction steering is done in parallel with register renaming. Because the `SRC_FIFO` table is smaller than the rename table, we expect the delay of steering to be less than the rename delay. In case a more complex steering heuristic is used, the extra delay can easily be moved into the wakeup/select stage, or a new pipestage can be introduced – at the cost of an increase in branch mispredict penalty.

In summary, the complexity analysis presented above shows that by reducing the delay of the window logic significantly, it is likely that the dependence-based microarchitecture can be clocked faster than the typical microarchitecture. In fact, from the overall delay results shown in Table 2, if the window logic (wakeup + select) is reduced substantially, register rename logic becomes the critical stage for a 4-way microarchitecture. Consequently, the dependence-based microarchitecture can improve the clock period by as much as (an admittedly optimistic) 39% in $0.18\mu m$ technology. Of course, this may require that other stages not studied here be more deeply pipelined. Combining the potential for a much faster clock with the results in Figure 13 indicates that the dependence-based microarchitecture is capable of superior performance relative to a typical superscalar microarchitecture.

5.4 Clustering the Dependence-based Microarchitecture

The real advantage of the proposed microarchitecture is for building machines with issue widths greater than four where, as shown by Table 2, the delay of both the large window and the long bypass busses can be significant and can considerably slow the clock. Clustered microarchitectures based on the dependence-based microarchitecture are ideally suited for such situations because they simplify both the window logic and the bypass logic. We describe one such microarchitecture for building an 8-way machine next.

[4] A column MUX is used to select the appropriate bit from each entry.

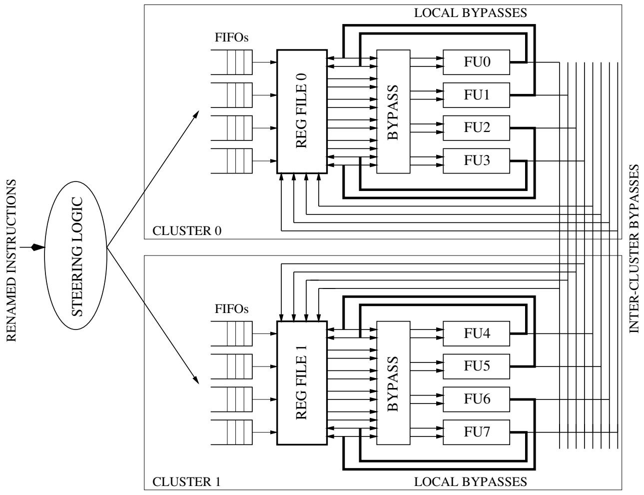

Figure 14: Clustering the dependence-based microarchitecture: 8-way machine organized as two 4-way clusters (2 X 4-way).

Consider the 2x4-way clustered system shown in Figure 14. Two clusters are used, each of which contains four FIFOs, one copy of the register file, and four functional units. Renamed instructions are steered to a FIFO in one of the two clusters. Local bypasses within a cluster (shown using thick lines) are responsible for bypassing result values produced in the cluster to the inputs of the functional units in the same cluster. As shown by the delay results in Table 2, local bypassing can be accomplished in a single cycle. Inter-cluster bypasses are responsible for bypassing values between functional units residing in different clusters. Because inter-cluster bypasses require long wires, it is likely that these bypasses will be relatively slower and take two or more cycles in future technologies. The two copies of the register file are identical, except for the one or more cycles difference in propagating results from one cluster to another.

This clustered, dependence-based microarchitecture has a number of advantages. First, wakeup and selection logic are simplified as noted previously. Second, because of the heuristic for assigning dependent instructions to FIFOs, and hence indirectly to clusters, local bypasses are used much more frequently than inter-cluster bypasses, reducing overall bypass delays. Third, using multiple copies of the register file reduces the number of ports on the register file and will make the access time of the register file faster.

5.5 Performance of Clustered Dependence-based Microarchitecture

The graph in Figure 15 compares performance, in terms of instructions committed per cycle (IPC), for the 2x4-way dependence-based microarchitecture against that of a conventional 8-way microarchitecture with a single 64-entry issue window. For the dependence-based system, instructions are steered using the heuristic described in Section 5.1 with a slight modification. Instead of using a single free list of empty FIFOs, we maintain two free lists of empty FIFOs, one per cluster. A request for a free FIFO is satisfied if possible from the *current* free list. If the current free list is empty, then the second free list is interrogated for a new FIFO and the second free list is made current. This scheme ensures that instructions adjacent in the dynamic stream are assigned to the same cluster to minimize inter-cluster communication. Local bypasses take one cycle while inter-cluster bypasses take 2 cycles. Also, in the conventional 8-way system, all bypasses are assumed to complete in a single cycle. From the graph we can see that for most of the benchmarks, the dependence-based microarchitecture is nearly as effective as the window-based microarchitecture even though the dependence-based microarchitecture is handicapped by slow inter-cluster bypasses that take 2 cycles. However, for two of the benchmarks, *m88ksim* and *compress*, the performance degradation is close to 12% and 9% respectively. We found that this degradation is mainly due to extra latency introduced by the slow inter-cluster bypasses.

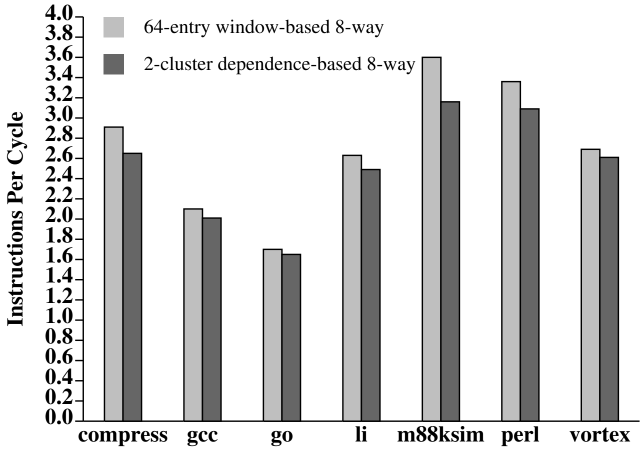

Figure 15: Performance of clustered dependence-based microarchitecture.

Because the dependence-based microarchitecture will facilitate a faster clock, a fair performance comparison must take clock speed into account. The local bypass structure within a cluster is equivalent to a conventional 4-way superscalar machine, and inter-cluster bypasses are removed from the critical path by taking an extra clock cycle. Consequently, the clock speed of the dependence-based microarchitecture is at least as fast as the clock speed of a 4-way, 32-entry window-based microarchitecture, and is likely to be significantly faster because of the smaller (wakeup + selection) delay compared to a conventional issue window as discussed in Section 5.3. Hence, if C_{dep} is the clock speed of the dependence-based microarchitecture, and C_{win} is the clock speed of the window-based microarchitecture, then from Table 2 for 0.18μm technology:

$$\frac{C_{dep}}{C_{win}} \geq \frac{delay\,of\;8\,way\;64\,entry\;window}{delay\,of\;4\,way\;32\,entry\;window} = 1.252$$

In other words, the dependence-based microarchitecture is capable of supporting a clock that is 25% faster than the clock of the window-based microarchitecture. Taking this factor into account (and ignoring other pipestages that may have to be more deeply pipelined), we can estimate the potential speedup with a dependence-based microarchitecture. The performance improvements vary from 10% to 22% with an average improvement of 16%.

5.6 Other Clustered Microarchitectures

The microarchitecture presented in the previous section is one point in the design space of clustered superscalar microarchitectures. The dependence-based microarchitecture simplifies both the window logic and naturally reduces the performance degradation due to slow inter-cluster bypass paths. In order to further explore the space, we studied the performance of some other interesting designs. In each case there are two clusters with inter-cluster bypasses taking an extra cycle to complete.

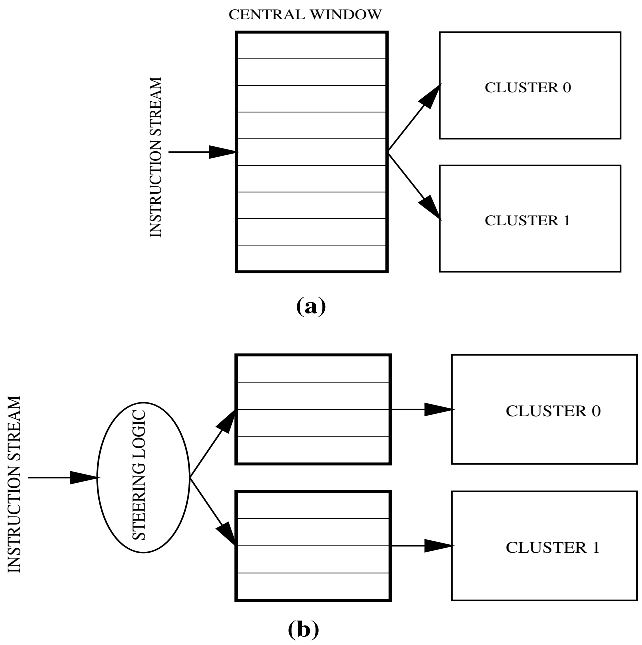

Figure 16: Clustered microarchitectures.

5.6.1 Single Window, Execution-Driven Steering

In the dependence-based microarchitecture described above, instructions are pre-assigned to a cluster when they are dispatched; we refer to this as *dispatch-driven* instruction steering. In contrast, Figure 16(a) illustrates a microarchitecture where instructions reside in a central window while waiting for their operands and functional units to become available. Instructions are assigned to the clusters at the time they begin execution; this is *execution-driven* instruction steering.

With this method, cluster assignment works as follows. The register values in the clusters become available at slightly different times; that is, the result register value produced by a cluster is available in that cluster one cycle earlier than in the other cluster. Consequently, an instruction waiting for the value may be *enabled* for execution one cycle earlier in one cluster than in the other. The selection logic monitors the instructions in the window and attempts to assign them to the cluster which provides their source values first (assuming there is a free functional unit in the cluster). Instructions that have their source operands available in both clusters are first considered for assignment to cluster 0. Static instruction order is used to break ties in this case.

The execution-driven approach uses a greedy policy to minimize the use of slow inter-cluster bypasses while maintaining a high utilization of the functional units. It does so by postponing the assignment of instructions to clusters until execution time. While this greedy approach may gain some IPC advantages, this organization suffers from the previously discussed drawbacks of a central window and complex selection logic.

5.6.2 Two Windows, Dispatch-Driven Steering

This microarchitecture, shown in Figure 16(b), is identical to the dependence-based clustered microarchitecture except that each cluster has a completely flexible window instead of FIFOs. Instructions are steered to the windows using a heuristic that takes both dependences between instructions and the relative load of the clusters into account. We tried a number of heuristics and found a simple extension of the FIFO heuristic presented in Section 5.1 to work best. In our scheme the window is modeled as if it is a collection of FIFOs with instructions capable of issuing from any slot within each individual FIFO. In this particular case, we treat each 32-entry window as eight FIFOs with four slots each. Note that these FIFOs are a conceptual device used only by the assignment heuristic – in reality, instructions issue from the window with complete flexibility.

Kemp and Franklin [11] studied an organization called PEWs (Parallel Execution Windows) for simplifying the logic associated with a central window. PEWs simplifies window logic by splitting the central instruction window among multiple windows much like the clustered microarchitecture described above. Register values are communicated between clusters (called pews) via hardware queues and a ring interconnection network. In contrast, we assume a broadcast mechanism for the same purpose. Instructions are steered to the pews based on instruction dependences with a goal to minimize inter-pew communication. However, for their experiments they assume that each of the pews has as many functional units as the central window organization. This assumption implies that the reduction in complexity achieved is limited since the wakeup and selection logic of the windows in the individual pews still have the same porting requirements as the central window.

5.6.3 Two Windows, Random Steering

This microarchitecture, using the structure presented in Figure 16(b), is a basis for comparisons. Instructions are steered randomly to one of the clusters. If the window for the selected cluster is full, then the instruction is inserted into the other (free) cluster. This design point was evaluated in order to determine the degree to which clustered microarchitectures are capable of tolerating the extra latency introduced by slow inter-cluster bypasses and the importance of dependence-aware scheduling. Each window has 32 entries in this case.

5.6.4 Performance of Clustered Microarchitectures

The top graph in Figure 17 shows the performance of various microarchitectures in terms of instructions committed per cycle (IPC). The leftmost bar in each group shows the performance of the ideal microarchitecture: a single 64-entry window with single cycle bypass between all functional units. A number of observations can be made from the figure. First, random steering consistently performs worse than the other schemes. The performance degradation with respect to the ideal case varies from 17% in the case of *vortex* to 26% in the case of *m88ksim*. Hence, it is essential for the steering logic to consider dependences when routing instructions. Second, the microarchitecture with a central window and execution-driven steering performs nearly as well as the ideal microarchitecture with a maximum degradation of 6% in the case of *m88ksim*. However, as discussed earlier in Section 5.6.1, this microarchitecture requires a centralized window with complex selection logic. Third, both the dependence-based microarchitecture and the flexible window microarchitecture using dispatch-driven steering perform competitively in comparison to the ideal microarchitecture.

The bottom graph in Figure 17 shows the frequency of inter-cluster communication for each organization. We measure inter-cluster communication in terms of the fraction of total instructions that exercise inter-cluster bypasses. This does not include cases where an instruction reads its operands from the register file in the cluster i.e. cases in which the operands arrived from the remote cluster in advance. As expected, we see that there is a high correlation between the frequency of inter-cluster communication and performance - organizations that exhibit higher inter-cluster communication commit fewer instructions per cycle. The inter-cluster communication is particularly high in the case of random steering, reaching as high as 35% in the case of *m88ksim*.

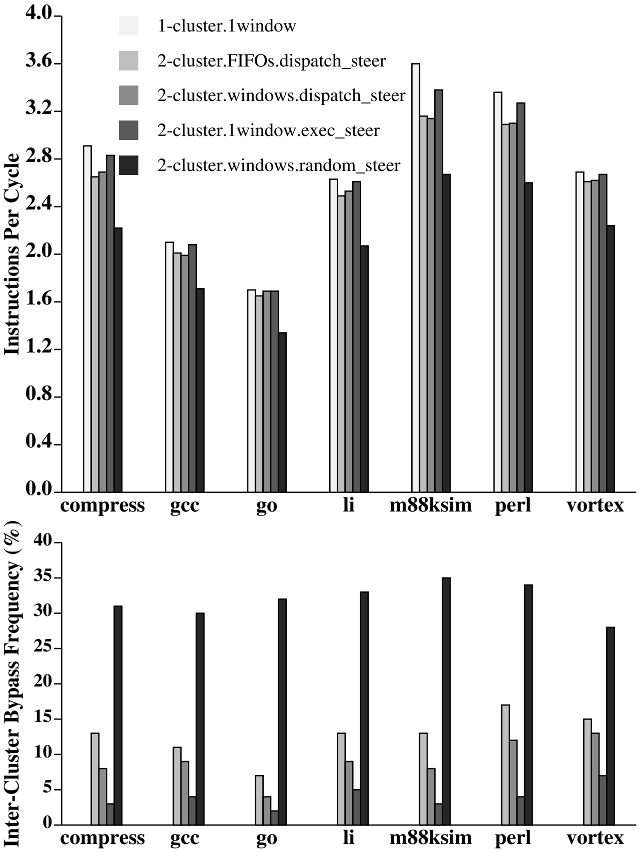

Figure 17: Performance of clustered microarchitectures.

Overall, the above results show that clustered microarchitectures using distributed windows coupled with dispatch-driven steering can deliver performance similar, in terms of instructions committed per cycle, to that of an ideal microarchitecture with a large window and uniform single cycle bypasses between all functional units.

6 Conclusions

We studied the variation of delays of key structures in a generic superscalar processor with two important microarchitectural parameters: issue width and issue window size. We also analyzed the impact of advanced technologies with smaller feature sizes on the delay of these structures. Our results show that the logic associated with the issue window and the data bypass logic are going to become increasingly critical as future designs employ wider issue widths, bigger windows, and smaller feature sizes. Furthermore, both of these structures rely on broadcasting values on long wires, and in future technologies wire delays will increasingly dominate total delay.

This is not to say that the delay of other structures, for example register files and caches, will not cause problems. However, these structures can be pipelined to some extent. In contrast, window logic and data bypass logic implement atomic operations that cannot be pipelined while allowing dependent instructions to execute in successive cycles. This characteristic makes the delay of the window logic and the data bypass logic even more crucial.

Hence, as architects build machines with wider issue widths and larger window sizes in advanced technologies, it is essential to consider microarchitectures that are complexity-effective i.e. microarchitectures that facilitate a fast clock while exploiting similar levels of ILP as an ideal large-window machine.

In the second half of the paper, we proposed one such microarchitecture called the dependence-based microarchitecture. The dependence-based microarchitecture detects chains of dependent instructions and steers the chains to FIFOs which are constrained to execute in-order. Since only the instructions at the FIFO heads have to be monitored for execution, the dependence-based microarchitecture simplifies window logic. Furthermore, the dependence-based microarchitecture naturally lends itself to clustering by grouping dependent instructions together. This grouping of dependent instructions helps mitigate the bypass problem to a large extent by using fast local bypasses more frequently than slow inter-cluster bypasses. We compared the performance of a 2x4-way dependence-based microarchitecture with a typical 8-way superscalar. Our results show two things. First, the proposed microarchitecture has IPC performance close to that of a typical microarchitecture (average degradation in IPC performance is 6.3%). Second, when taking the clock speed advantage of the dependence-based microarchitecture into account the 8-way dependence-based microarchitecture is 16% faster than the typical window-based microarchitecture on average.

Acknowledgements

This work was supported in part by an internship at DEC Western Research Laboratory, and by grants from the NSF Grant MIP-9505853, and the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

We thank the staff at DEC-WRL, especially Annie Warren and Jason Wold, for providing us with the CAD tools used in this study. Thanks to Andy Glew for answering specific questions regarding the implementation of the Intel Pentium Pro and for related discussions. Thanks also to Todd Austin, Scott Breach, and Shamik Das Sharma for comments on a draft of this paper.

References

[1] P. S. Ahuja, D. W. Clark, and A. Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[2] C. Asato, R. Montoye, J. Gmuender, E. W. Simmons, A. Ike, and J. Zasio. A 14-port 3.8ns 116-word 64b Read-Renaming Register File. In *1995 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 104–105, February 1995.

[3] Mark T. Bohr. Interconnect Scaling - The Real Limiter to High Performance ULSI. In *1995 International Electron Devices Meeting Technical Digest*, pages 241–244, 1995.

[4] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-96-1308 (Available from http://www.cs.wisc.edu/trs.html), University of Wisconsin-Madison, July 1996.

[5] M. Butler and Y. N. Patt. An Investigation of the Performance of Various Dynamic Scheduling Techniques. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 1–9, December 1992.

[6] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register File Design Considerations in Dynamically Scheduled Processors. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[7] Linley Gwennap. Speed Kills? Not for RISC Processors. *Microprocessor Report*, 7(3):3, March 1993.

[8] Linley Gwennap. HAL Reveals Multichip SPARC Processor. *Microprocessor Report*, 9(3), March 1995.

[9] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2), February 1995.

[10] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, October 1996. 9th Annual Microprocessor Forum, San Jose, California.

[11] Gregory A. Kemp and Manoj Franklin. PEWs: A Decentralized Dynamic Scheduler for ILP Processing. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 239–246, 1996.

[12] Ashok Kumar. The HP-PA8000 RISC CPU: A High Performance Out-of-Order Processor. In *Proceedings of the Hot Chips VIII*, pages 9–20, August 1996.

[13] Scott McFarling. Combining Branch Predictors. DEC WRL Technical Note TN-36, DEC Western Research Laboratory, 1993.

[14] Meta-Software Inc. *HSpice User's Manual*, June 1987.

[15] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-96-1328 (Available from http://www.cs.wisc.edu/trs.html), University of Wisconsin-Madison, November 1996.

[16] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 RISC Microprocessor. In *IEEE Micro*, pages 8–17, October 1995.

[17] N. Vasseghi et al. 200 MHz Superscalar RISC Processor Circuit Design Issues. In *1996 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 356–357, February 1996.

[18] Tomohisa Wada, Suresh Rajan, and Steven A. Przybylski. An Analytical Access Time Model for On-Chip Cache Memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992.

[19] Neil H.E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, second edition, 1993.

[20] Neil C. Wilhelm. Why Wire Delays Will No Longer Scale for VLSI Chips. Technical Report SMLI TR-95-44, Sun Microsystems Laboratories, August 1995.

[21] Steven J. E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, DEC Western Research Laboratory, July 1994.

[22] K. C. Yeager. MIPS R10000 Superscalar Microprocessor. In *IEEE Micro*, April 1996.