

Ultrascalar Memo 5

# Circuits for Wide-Window Superscalar Processors<sup>1</sup>

Dana S. Henry, Bradley C. Kuszmaul, Gabriel H. Loh, and Rahul Sami  
Yale University Departments of Computer Science and Electrical  
Engineering  
{dana,bradley,lohg,samir}@eecs.yale.edu

June 2000

## Abstract

Our program benchmarks and simulations of novel circuits indicate that large-window processors are feasible. Using our redesigned superscalar components, a large-window processor implemented in today's technology can achieve an increase of 10–60% (geometric mean of 31%) in program speed compared to today's processors. The processor operates at clock speeds comparable to today's processors, but achieves significantly higher ILP.

To measure the impact of a large window on clock speed, we design and simulate new implementations of the logic components that most limit the critical path of our large-window processor: the schedule logic and the wake-up logic. We use log-depth cyclic segmented prefix (CSP) circuits to reimplement these components. Our layouts and simulations of critical paths through these circuits indicate that our large-window processor could be clocked at frequencies exceeding 500MHz in today's technology. Our commit logic and rename logic can also run at these speeds.

To measure the impact of a large window on ILP, we compare two microarchitectures, the first has a 128-instruction window, an 8-wide fetch unit, and 20-wide issue (four integer, branch, multiply, float, and memory units), whereas the second has a 32-instruction window, and a 4-wide fetch unit and is comparable to today's processors. For each, we simulate different window reuse and bypass policies. Our simulations show that the large-window processor achieves significantly higher IPC. This performance increase comes despite the fact that the large-window processor uses a wrap-around window while the small-window processor uses a compressing window, thus effectively increasing its number of outstanding instructions. Furthermore, the large-window processor sometimes pays an extra clock cycle for bypassing.

## 1 Introduction

It is so difficult to design a high-speed wide-issue superscalar processor that some processor makers seem to be abandoning the whole idea. The problem appears to be that the logic to decode, rename, analyze, and schedule  $n$  instructions per clock cycle slows the clock cycle down enough to result in a net performance decrease compared to a processor that issues fewer instructions per clock. Examples of this trend include IBM's Power4, which includes two 4-issue processors on a chip instead of a single wider-issue processor, and Intel's Itanium which relies on VLIW techniques to reduce the amount of analysis and scheduling done at runtime. Countering this trend is Compaq's announcement that the Alpha EV8 will include a window of 256 instructions and a peak issue rate of 8 instructions per clock with support for four threads. It is not yet clear what performance characteristics the EV8 will have. To give an example of the sort of performance we mean, consider the Alpha 21264 (EV6), which uses two small windows (20 entries for integer and 15 for float) instead of one big window (see [2] for a description of the issue logic in the EV6.) The integer window statically assigns each instruction to a group of functional units before enqueueing it. It requires

<sup>1</sup>This work was partially supported by NSF Career Grants CCR-9702980 (Kuszmaul) and MIP-9702281 (Henry) and by an equipment grant from Intel.

Appeared in *The International Symposium on Computer Architecture (ISCA)*, pp. 236–247, June 12–14, 2000, Vancouver, British Columbia.

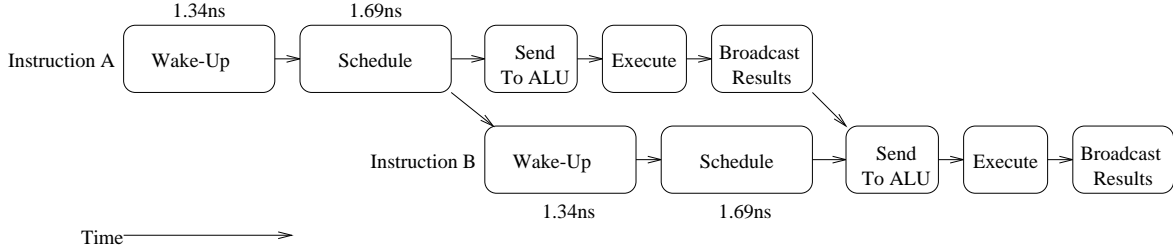


Figure 1: The steps taken to execute two dependent arithmetic instructions and their dependencies.

an extra clock cycle for data to move between instructions that happen to have been placed far apart from each other, as compared to if they had been placed near each other. The collective effect is that the EV6 is already paying for its large window size (although the overall cost is apparently acceptable—perhaps 2% on SPEC benchmarks.) We would expect the EV8 to pay relatively more since its window is bigger and it has more functional units. The EV8 will gain a tremendous advantage over today’s technology by using a copper/low-dielectric-constant SOI process in the year 2003, making it difficult to compare to our study which relies on a  $0.25\mu\text{m}$  aluminum technology that is available today. (The Power4, Itanium, and EV8 were all described at the 1999 Microprocessor Forum.)

This paper outlines the core of a processor that can fetch 8 instructions per clock, issue 20 instructions per clock, and has a window of 128 instructions. This processor, designed in the technology of mid 1999 ( $0.25\mu\text{m}$  aluminum), has critical path competitive to today’s processors (our critical path is under 2ns) and with substantially higher ILP and program speed compared to today’s processors. Our processor relies on a novel design of the wake-up logic and of a multi-unit scheduler. Our designs enable cyclic reuse of the reordering buffer with new instructions continually entering the buffer and taking up the place of the oldest, retiring ones, without having to use circuitry to compress instructions to the beginning of the reordering buffer.

We have concentrated on redesigning the processor components that limit the execution time of dependent arithmetic instructions in the reordering buffer. Figure 1 shows the steps that must be taken in order to execute two dependent arithmetic instructions without bypassing. In our example, Instruction B depends on the result of Instruction A. Instruction A wakes up Instruction B, once A has been successfully scheduled. Instruction B requests to be scheduled while waiting for the result of A. According to SPICE simulations of our layouts, our wakeup logic runs in 1.34ns and our scheduler logic runs in 1.69ns.

Our circuit designs should be viewed as only one stake in the ground. Earlier study of the MIPS R10000 and the Alpha 21264 showed that their circuit implementations of superscalar components would not scale to large buffer sizes [9]. Subsequent processors, such as the AMD K6, have begun to use more scalable implementations to reimplement some of these components. There may well be other, possibly better, designs for the processor components described in this paper. To our knowledge, no such designs have been published.

While we present new scalable designs for some processor components in this paper, there are many other processor components that we have not addressed. We have not redesigned the processor’s data paths, only the control paths. We have also not redesigned the logic for bypassing results among numerous functional units. Instead, in our program performance study, we measure a system with no bypasses. Finally, we have not addressed the problems of scaling the memory subsystem. In our program study, we assume a 32-entry memory buffer that has comparable functionality to the Alpha 21264’s buffer. While we believe that such a buffer is feasible, we have not designed it.

All of our redesigned superscalar components draw on the same underlying idea. They all exploit the sequential ordering of instructions in a wrap-around reordering buffer and attach one or more *cyclic segmented prefix (CSP)* circuits to the reordering buffer. Figure 2(a) illustrates an eight-instruction wrap-around reordering buffer. Instructions are stored in the buffer in a wrap-around sequence. The oldest instruction in the buffer is Instruction A, pointed to by the *Head* pointer. The youngest, most recently fetched, is Instruction H pointed to by the *Tail* pointer.

This work was partly motivated by our previous theoretical results on asymptotically optimal superscalar processors [3, 6]. In contrast, this work focuses on understanding the engineering problems of the wide-issue processors of the near future.

Figure 2(a) also shows a linear gate-delay implementation of a CSP circuit. A CSP circuit with a linear gate delay consists of a ring of operators,  $\otimes$ , and MUXes. We attach this ring to the wrap-around reordering buffer using different associative operators,  $\otimes$ . The  $j$ th entry in the buffer is attached to input  $in_j$ , output  $out_j$ , and segment bit  $s_j$  of the CSP

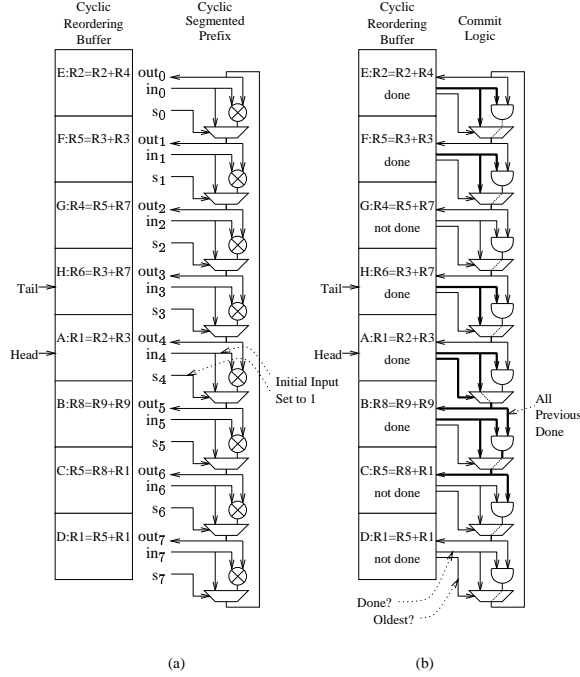


Figure 2: (a) An 8-entry wrap-around reordering buffer with adjacent, linear gate-delay cyclic segmented prefix (CSP). The  $\otimes$  can be any associative operator, (b) Commit logic using CSP.

circuit. The circuit applies the operator  $\otimes$  to successive inputs and assigns the result accumulated so far, also known as a prefix, to each output. The circuit stops accumulating whenever it encounters a high segment bit. For example, if  $s_6 = 1$  and  $s_7 = s_0 = s_1 = 0$ , then  $out_2 = in_6 \otimes in_7 \otimes in_0 \otimes in_1$ . For the circuit to produce well-defined values, at least one instruction, typically the oldest, must set its segment bit high in order to stop the cyclic accumulation of inputs. In general, many instructions can raise their segment bits, leading the circuit to accumulate inputs over multiple non-overlapping, adjacent segments. Although Figure 2(a) shows a linear gate-delay implementation of a CSP circuit; other, logarithmic gate-delay implementations exist. Figures 4(a), 4(b), 5, and 6 illustrate four such implementations. All the CSP implementations have identical interfaces and functionality, but the logarithmic gate-delay implementations can lead to dramatically faster circuits.

The rest of this paper describes our novel circuits, their VLSI layouts, and simulations, and analyzes the benefits of a large-window processor utilizing these circuits. Section 2 describes our designs of the wakeup, schedule, commit, and rename logic in terms of linear gate-delay CSP circuits. Section 3 converts linear gate-delay CSP circuits to faster, logarithmic gate-delay CSP circuits and compares several alternative designs. Section 4 describes and analyzes our VLSI implementations of wakeup, schedule, and commit logic. Section 5 describes our program performance study and analyzes its results. Section 6 discusses implications for building a wide-window processor in future technologies.

## 2 CSP Circuits for Superscalar Components

This section shows how different superscalar components can be redesigned using CSP circuits. Using CSP circuits, we redesign the commit logic, the wakeup logic, the schedule logic, the rename logic, and the commit logic of a traditional superscalar processor. To simplify our explanation, we show each component redesigned with linear gate-delay CSP circuits first. In Section 3, we will convert our designs to faster logarithmic gate-delay CSP circuits.

Consider, first, the commit logic. The commit logic informs each instruction whether all earlier instructions in the buffer have committed. Figure 2(b) shows a linear gate-delay implementation of the commit logic attached to our eight-instruction wrap-around reordering buffer. The commit logic consists of a single one-bit-wide CSP circuit with the operator AND. The AND gates accumulate the successive answer: “Have all earlier instructions committed?”

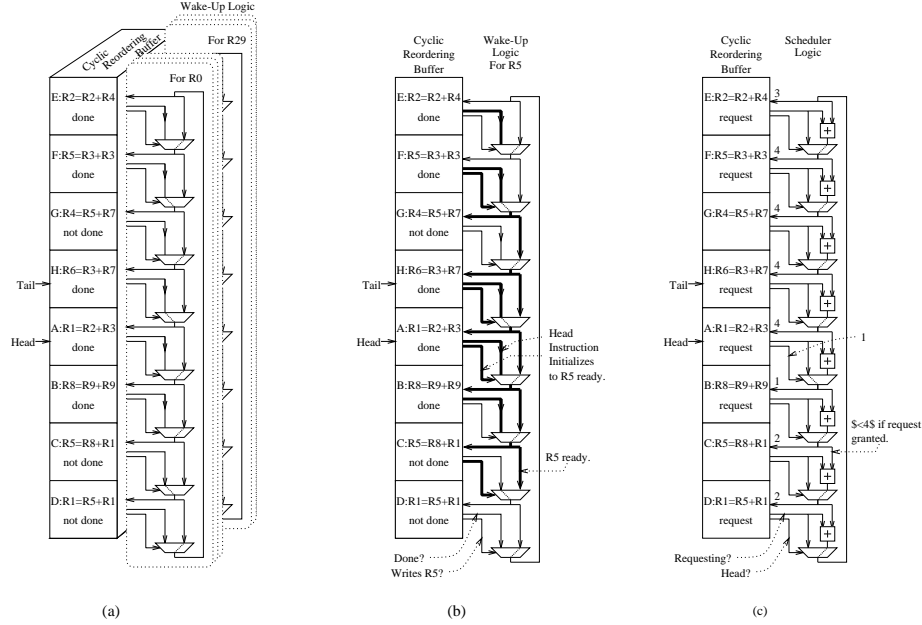


Figure 3: (a) An 8-entry wrap-around reordering buffer with adjacent wake-up logic for a processor with 32 logical registers. (b) The wake-up logic for logical register R5. Asserted signals are shown in bold. (c) Scheduler logic scheduling four functional units.

Each multiplexer passes the accumulated answer to successive instructions, but stops at the oldest one.

Figure 2(b) includes an example. In the example, wires carrying high signals are displayed in bold; Instructions A, B, E, and F have committed; and the commit logic has informed Instructions B and C that all earlier instructions have committed. Instructions A, B, and C can now act based on the output of the commit logic and their own status. Instructions A and B retire, while Instruction C becomes the new Head Instruction.

Our wake-up logic uses CSP circuits to determine when each instruction’s arguments are ready to be latched off a broadcast bus. Latched arguments remain in the window entry until the entry can be scheduled. The wake-up logic uses one CSP circuit for each logical register defined in the processor’s instruction set architecture. Each CSP circuit operates independently of the others and informs the buffer’s instructions about the readiness of its logical register. Figure 3(a) shows our wake-up logic for a processor with 32 logical registers. Each instruction in the reordering buffer receives 32 ready bits indicating the readiness of each register. Each instruction then uses a 32-to-1 multiplexer (not shown), for each of its arguments, to select the ready bits corresponding to the registers it needs.

Figure 3(a) illustrates our linear gate-delay implementation of the wake-up logic for one register, register R5. The figure shows the values passing along the wake-up CSP circuit. Wires carrying high signals are displayed in bold. The operator  $\otimes$  for this CSP circuit is simply a wire that passes the old value along (i.e.,  $a \otimes b = a$ ). Each instruction in the reordering buffer sets its segment bit high if it writes register R5. It sets its input bit high once it has computed R5’s value. In our figure, Instruction F has already computed a value of R5 and set its input bit high; Instruction C has not. As a result, Instruction G is informed that R5 is ready, but Instruction E is not.

Our schedule logic uses a single CSP circuit with addition for its operator  $\otimes$ . Figure 3(c) illustrates our scheduler which assigns four functional units to the four oldest requesting instructions in a wrap-around reordering buffer. For each buffer entry, the scheduler simply returns the sum,  $n$ , of all the older instructions requesting to be scheduled. (The sum can saturate at the number of functional units.) A requesting entry is scheduled to use functional unit  $n$ , if  $n$  is less than the number of functional units. In the example of Figure 3(c), Instructions A, B, D, and E have been scheduled to functional units 0, 1, 2, and 3 respectively.

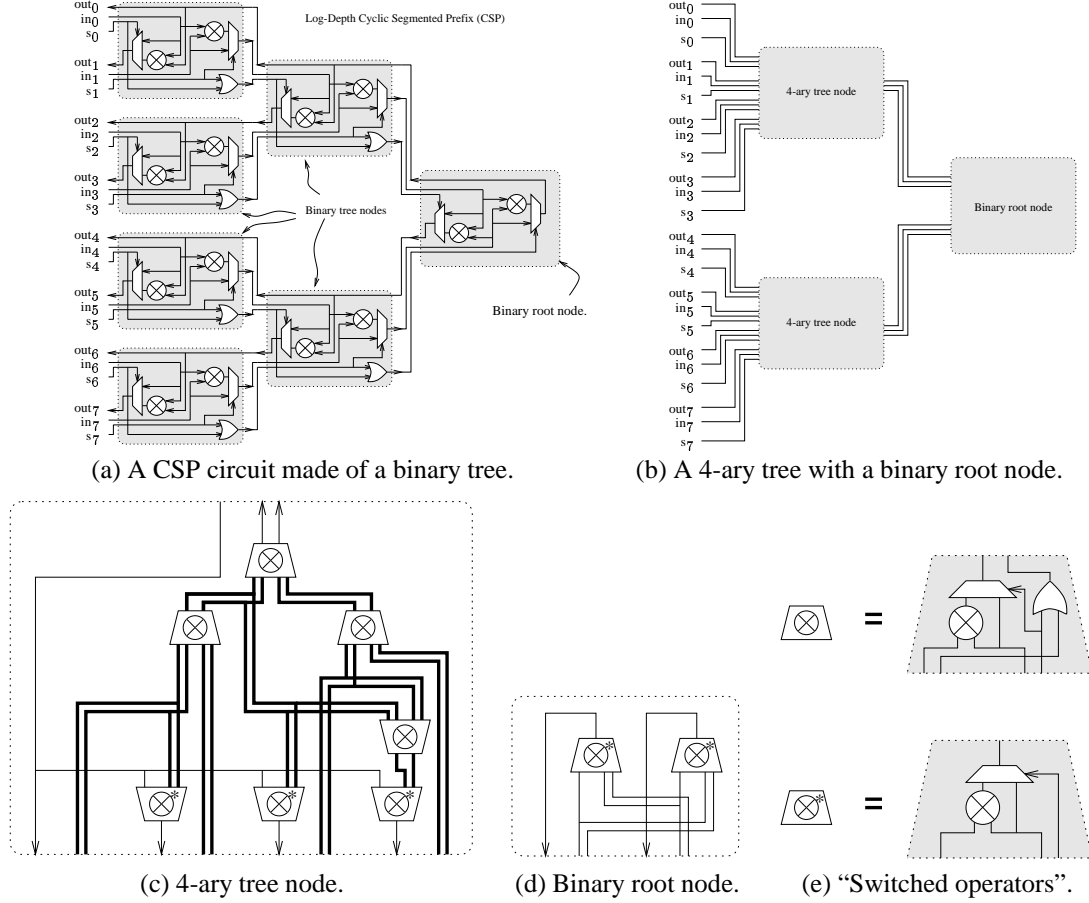


Figure 4: (a) A binary-tree implementation of CSP. (b) A 4-ary tree with a binary root node. (c) A node in the 4-ary tree. (d) The binary root node. (e) The switched operators used in (c) and (d).

### 3 Alternative CSP Circuits

Although, for simplicity, the figures above show linear gate-delay prefix circuits, we found that logarithmic gate-delay implementations can significantly reduce the critical path delay. This section describes and contrasts four different implementations of CSP circuits that all have only logarithmic gate delay in the window size. The next section will discuss the simulations and the layouts of our superscalar components built from these CSP circuits.

All four CSP circuits described in this section implement the same function as the circuit of Figure 2(a). While the linear gate-delay CSP circuit in Figure 2(a) applied the  $\otimes$  operator in-order to successive inputs, the four logarithmic gate-delay CSP circuits rely on the associativity of the  $\otimes$  operator, by applying the operator in parallel to contiguous subsets of the inputs. They all have  $O(\lg n)$  delays due to gates, but they have varying areas and delays due to wires. The four circuits are: a binary tree, a 4-ary tree, a “thicket” of trees and a “prefix/postfix thicket”.

The binary tree circuit is shown Figure 4(a). The binary tree consists of a collection of binary tree nodes (each shown with grey backgrounds) that compute a segmented prefix in the way described in [1]. Our circuit is different from the circuit of [1] in that we modified the root node to make the tree implement a *cyclic* segmented prefix instead of an acyclic segmented prefix. For a reordering buffer with  $n$  instructions, the gate delay through the binary tree implementation consists of  $((2 \lg n) - 1) \otimes$  operator delays<sup>2</sup> plus  $((2 \lg n) - 1)$  MUX delays. The delays can be thought of as  $(\lg n - 1)$  operators and MUXes going up the tree, followed by  $\lg n$  operators and MUXes going down the tree.

A faster version of the tree circuit can be implemented by building a 4-ary tree, as shown in Figure 4(b). The details

<sup>2</sup>We write  $\lg n$  to for the log base 2 of  $n$ .

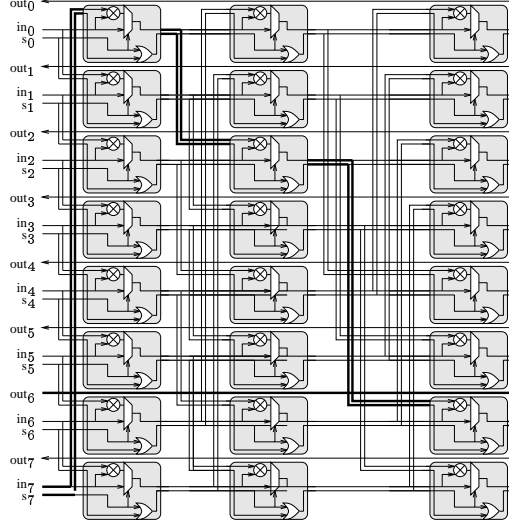


Figure 5: A CSP circuit made of a wrap-around “thicket”, with one longest path highlighted.

of a 4-ary tree node are shown in Figure 4(c–e). The binary root node shown in Figure 4(d) is the same one used in Figure 4(a). The delays going up the 4-ary tree are the same as the binary tree (although, as we shall see in Section 4, using a compound gate to implement a 4-ary MUX can speed up the circuit further.) The delays going down the tree are halved, however. This is because the values going up the tree arrive first and precompute all the values shown in bold. Later, when the value coming down the tree finally arrives, it passes through only one switched operator at the bottom of the 4-ary tree node. Thus the gate delay consists of only  $\lfloor 3/2 \lg n \rfloor \otimes$  operator delays and  $\lfloor 3/2 \lg n \rfloor$  MUX delays. The use of 4-ary trees to implement acyclic prefix is well known (see, for example, the scheduler logic in [9]), but we have not seen any 4-ary trees that implement a cyclic prefix.

The 4-ary tree idea can be generalized to other widths. For example, whereas a 4-ary node produces a circuit with a gate delay of  $\lfloor 3/2 \lg n \rfloor \otimes$  operator and MUX delays, an 8-ary node produces a circuit with only  $\lfloor 4/3 \lg n \rfloor \otimes$  operator and MUX delays.

The third approach is to build a “thicket” of trees, such as is described in [1, Exercise 29.2-6]. Figure 5 illustrates this method. Once again, the main difference between this circuit and the one in the literature is that our circuit implements a *cyclic* prefix operation. The gate delay through a thicket implementation consists of only  $\lg n \otimes$  operator and MUX delays. The area increases substantially, however; and the savings in gate delay are partly offset by increased wire delays to traverse that area.

One disadvantage of the thicket is that some signals must travel all the way from the bottom of the circuit to the top of the circuit and then all the way back down. Consider, for example, a scenario in which all segment bits are low except for  $s_6$ , the next-to-last window entry. Figure 5 highlights one resulting path through the circuit. The value from the last window entry must travel all the way to the top of the circuit in the first stage, and then work its way nearly to the bottom of the circuit in the subsequent stages.

To address this doubled-wire-length problem in the thicket circuit, we developed what we call a “prefix-postfix thicket”. The prefix-postfix thicket, shown in Figure 6, combines the outputs of an acyclic, segmented prefix and an acyclic, segmented postfix in order to generate a CSP. For example, Figure 6 highlights the datapath that computes  $out_3$ , assuming that only one segment bit,  $s_5$ , is high. The prefix circuit computes  $in_0 \otimes (in_1 \otimes in_2)$ . The postfix circuit computes  $(in_5 \otimes in_6) \otimes in_7$ . Since the prefix’s segment bit is low, the root node combines the outputs of the prefix and the postfix circuits, in the correct order, generating the answer:

$$out_3 = ((in_5 \otimes in_6) \otimes in_7) \otimes (in_0 \otimes (in_1 \otimes in_2)).$$

As our example illustrates, the “prefix-postfix thicket” computes any output signal while traversing the height of the reordering buffer at most once.

The thickets requires much more area than do the trees. The tree’s area grows as  $\Theta(n \lg n)$  since the height of the layout is  $\Theta(n)$  and the width of the layout is  $\Theta(\lg n)$ . (An H-tree layout could get the area of a tree down to

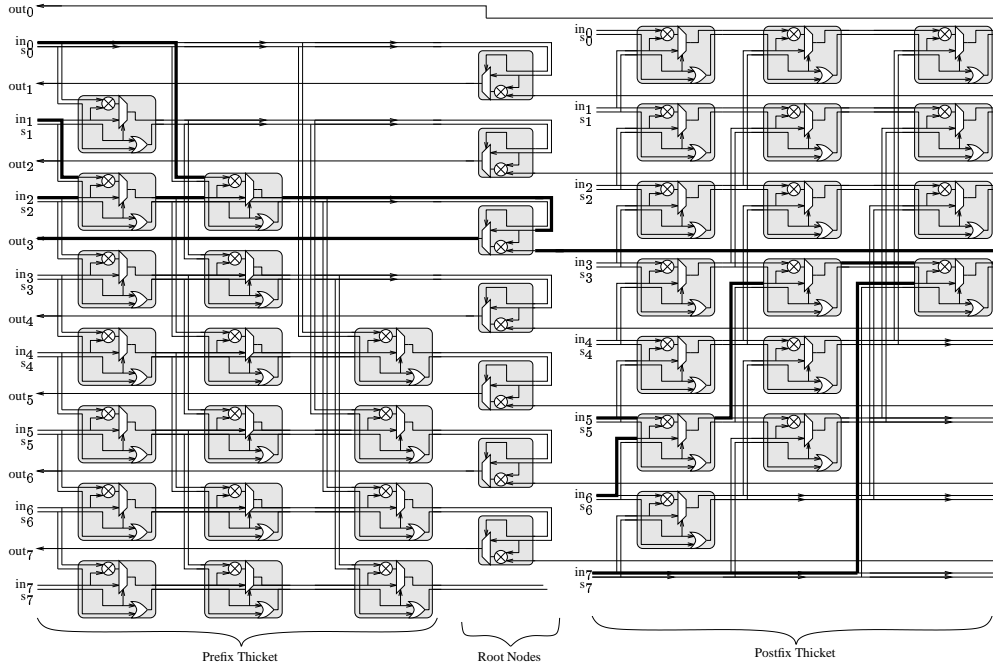


Figure 6: A CSP circuit made of an acyclic prefix thicket, an acyclic postfix thicket, and root nodes that combine the results from the two acyclic thickets. The wires used by one particular reduction is shown with thick lines.

$\Theta(n)$ , but we are assuming that the window entries are laid out in a linear array for this study.) The thicket's area grows as  $\Theta(n^2)$  for  $n$  window entries, since the height of the layout is  $\Theta(n)$ , and the width of the last stage alone is  $\Theta(n)$  because  $n/2$  wires must move from the bottom half of the circuit to the top half. The thicket has the advantage, however, that it has half the gate delays of the binary tree. The 4-ary tree is somewhere in between with slightly greater area and about  $3/4$  the gate delays of the binary tree.

## 4 Implementation and Performance

Having enumerated several logarithmic-depth prefix circuits, we will next describe and evaluate our VLSI implementations of wake-up, schedule, commit, and rename logic using each of these circuits and compare the different implementations.

To avoid having a very long thin reordering buffer, we assume in our implementations that the reordering buffer is laid out in two columns of 64 buffer entries each (see Figure 7.) Each buffer entry is assumed to be  $1000\lambda$  high. We believe that  $1000\lambda$  is an overestimate of the height, possibly by more than a factor of two. We decided to use a larger-than-necessary buffer height so that our critical-path-length estimate would be too high rather than too low. The various broadcast circuits connecting the window entries to functional units run vertically over the entries, while the commit, wake-up, and schedule circuits run between the two columns of entries. Our circuits' wire lengths reflect this layout. They include vertical wire lengths to traverse the height of reordering buffer as well as horizontal wire lengths to traverse the other circuits sandwiched between the two columns.

Having settled on the buffer's layout, we then designed our superscalar components from Section 2 using each of the CSP implementations from Section 3. We based our designs on a  $0.25\mu\text{m}$ , 5-metal-layer, Aluminum CMOS technology. For each design, we considered a number of alternative implementations, in static, domino and transmission gate logic. We also considered many different sizes for each gate along each circuit's critical path. We did not consider more than one size ratio for transistors within the same compound gate, however. Different size ratios could perhaps yield circuits faster than the ones we report.

We have sized the transistors of our circuits for maximum speed, using a C program that we wrote. Our program assumes that the inputs are minimum-sized and includes any needed step-up inverters. Outputs drive minimum sized

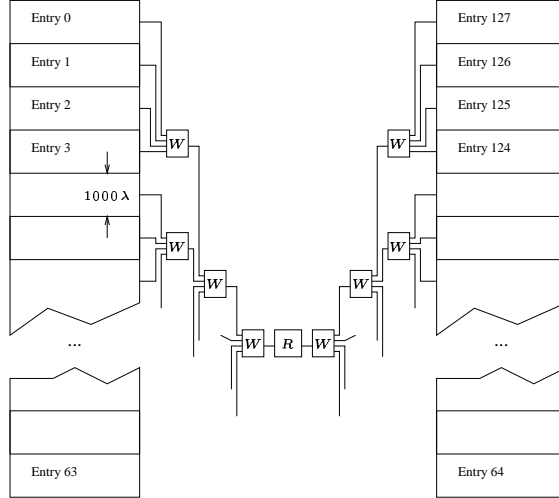


Figure 7: The layout of the register window with the 4-ary wakeup logic tree shown.

	2-Tree		Area (Mλ <sup>2</sup> )	4-tree		Area (Mλ <sup>2</sup> )	Thicket		Area (Mλ <sup>2</sup> )	Pre/Post		Area (Mλ <sup>2</sup> )
	Delay (ns) Est'd	SPICE		Delay (ns) Est'd	SPICE		Delay (ns) Est'd	SPICE		Delay (ns) Est'd	SPICE	
Commit	2.06		9	1.82		11	1.76		140	1.46	<b>1.41</b>	160
Wakeup	2.35		900	1.60	<b>1.54</b>	980	1.99		13000	1.68	1.80	15000
(T-gates)	1.78	1.94	770	1.48	<b>1.34</b>	800						
Schedule	2.35		24	2.10		26	2.03		320	1.64	<b>1.69</b>	360

Figure 8: Circuit delays and areas including step-up and wire costs. Circuits in all the rows, except for the “T-gates” row, use domino logic. Circuits in the “T-gates” row use transmission gates.

gates. The program models the delay of a transistor and of a wire by a piecewise-quadratic approximation function that we fitted to match the SPICE simulations of the 0.25μm technology [18]. The program’s input consists of a set of allowed sizes, and a gate-level description of the circuit’s critical path annotated with wire lengths. The program starts out by assigning a random size to each gate, computes the critical path’s delay, and then iterates using a genetic search algorithm to reassign sizes. It takes the program only a few minutes to converge to a very good circuit.

We used our estimates to choose the most promising circuits and implement them. We layed out the critical paths of these circuits in Magic, and extracted the circuits using a model which distributes the RC of long wires into a series of resistors and capacitors. We ran SPICE on the circuits, and found that our sizing program’s estimates of the delays are consistently within 10% of the SPICE results.

Figure 8 summarizes, in a table, the results for our best designs. The figure assumes a processor with 32 logical registers and 128-entry wrap-around reordering buffer layed out in two columns. The processor’s wake-up logic includes all 32 CSP circuits plus a 32-to-1 multiplexer for each argument in the reordering buffer. The processor’s schedule logic assigns four functional units to the four oldest requesting instructions in the reordering buffer. Each column of the table uses a different CSP implementation from Section 3. Most of the circuits in the table are implemented with domino logic and driving inverters. There are two exceptions. The 32-to-1 MUXes within our wake-up logic are implemented with transmission gates. And the row labeled “T-gates” describes faster implementations of wake-up logic in which all multiplexers are built with transmission gates.

The table shows the critical path delay and area estimate for our best commit, wake-up, and schedule designs. For all designs, the table reports the estimated delays generated by our sizing program. The table also includes the delays reported by SPICE for the circuits’ critical paths that we have layed out in Magic and simulated in SPICE. Finally, the table gives an estimate of each component’s area that accounts for both gates and wires. Our area estimates use four metal layers to route signals. Using only four layers likely overestimates area, since existing aluminum technologies already use eight metal layers. Our area estimates may also be somewhat high because we have not optimally sized

gates outside of critical paths <sup>3</sup>.

The remainder of this section describes our implementations of each logic component in greater detail and analyzes their performance.

### Wake-Up Logic

The description of the wakeup logic in Section 1 presented a simplified view. Our wake-up logic does not only compute the readiness of each argument. It also propagates the number of the functional unit producing each result. A woken-up instruction can use the functional unit number to read its argument off the unit's result bus. The actual prefix circuit that we simulated thus passes 5-bit values (a ready bit, plus four bits identifying one of sixteen result-generating functional units) through the multiplexers. Thus the circuit is the same as the CSP circuit described in Figure 3, but values traveling through the prefix are 5 bits wide instead of 1 bit wide.

Figure 8 confirms that the wake-up logic is the most area-intensive of our components. Fortunately, one of the least area-intensive implementations, the 4-ary tree, is also the fastest. The resulting wake-up logic's width, for all 32 logical registers, is less than one fourth of the height of the two-column buffer.

Using transmission gates, rather than domino logic to implement each multiplexer within the wake-up logic can further speed up the design. We have sized, layed out, and simulated with SPICE the wake-up logic's critical path, using transmission gates to implement MUXes and trees to implement CSP's. Each tree node consists only of transmission-gate multiplexer(s) and driving inverters. The binary tree implementation runs in 1.94ns, whereas the 4-ary tree implementation runs in only 1.34ns according to our SPICE simulation. The 4-ary tree implementation speeds up much more than the binary tree implementation because a 4-ary transmission gate MUX is almost as fast as a 2-ary one, when the select bits are ready in advance.

### Scheduler Logic

Our scheduler schedules four functional units as illustrated in Figure 3. All 128 reordering buffer entries can request a unit. The four oldest requesters receive positive acknowledgements together with the number of the unit that has been assigned to them. The rest receive a negative acknowledgement.

Not surprisingly, the scheduler is our slowest component. To minimize delay, we have layed out the critical path of our scheduler using a prefix/postfix thicket implementation of CSP and a unary encoding of each sum propagating through the thicket. SPICE simulations of our critical path yielded worst-case delay of 1.69ns, whereas our sizing program predicted 1.64 ns.

In our study of instruction-level-parallelism, we use five separate schedulers to schedule four integer ALUs, four branch units, four memory units, four integer multiply units, and four floating point units. The five schedulers implemented with prefix/postfix thickets, require more total area than our wake-up logic implemented with a 4-ary tree. Together, the five schedulers' width is less than half of the height of the the two-column reordering buffer. We account for this width when computing the wire delays of all of our circuits.

Our scheduler's speed compares favorably to the one described by Palacharla [10, 9]. Palacharla used a synthetic 0.35 $\mu\text{m}$  and 0.18 $\mu\text{m}$  process extrapolated from 0.8 $\mu\text{m}$  and 0.5 $\mu\text{m}$  technologies. We used 0.25 $\mu\text{m}$  process parameters from MOSIS. Palacharla did not account for wire delays, whereas we did. We used a linear interpolation of the delay between the 0.35 $\mu\text{m}$  and 0.18 $\mu\text{m}$  to conclude that Palacharla predicts scheduler delays of about 0.8ns for a window of 128. In comparison our scheduler circuit for one functional unit has a delay of 0.71ns if we assume that all wires are of length 0. If we assume that the windows are 100 $\lambda$  high, then our circuit incurs a delay of 0.88ns, and if we assume that the windows are 1000 $\lambda$  high, then our circuit is 1.32ns.

Palacharla shows how to schedule two functional units of the same type (e.g., two FP adders), by chaining two schedulers together, which would give a delay of 2.64ns using 1000 $\lambda$  windows. Our scheduler for twice as many functional units (four instead of two) with windows of 1000 $\lambda$  has a delay of only 1.69ns.

### Commit Logic

We have layed out the critical path through a prefix/postfix thicket which computes the commit bits within 1.41ns. Since the commit CSP is only one bit wide, the VLSI area of the prefix/postfix thicket is negligible.

### Rename Logic

If each instruction in our instruction-set architecture generates only one result, we can implement rename logic in much the same way as our wake-up logic. We pass through each logical register's CSP a 7-bit address instead of a

<sup>3</sup>Wire area dominates gate area, however, limiting the possible overestimate to at most 20–25%.

4-bit functional-unit number and a ready bit. The rename logic supplies each entry in the entire buffer with the physical register numbers of its arguments. The physical register numbers in this implementation are the reordering buffer addresses of the instructions that write them.

## 5 Performance Impact

The preceding sections show a strategy for redesigning many of the superscalar components to operate on a large wrap-around reordering buffer. Our circuits implement a reordering buffer that is more than three times larger than the reordering buffers of today’s commercial processors, while reaching comparable clock speeds in a comparable technology. Program performance does not just depend on clock speeds, however, but also on the instruction-level parallelism (ILP) uncovered by the processor. In this section, we will try to quantify the effect larger reordering buffers might have on the ILP of next-generation processors.

### Our Simulation Environment

We based our studies of ILP on the SIMOS instruction-level simulator of the Alpha Instruction Set Architecture [13, 17]. To measure instruction-level parallelism, we added a time-stamping mechanism to the SIMOS in-order simulator. We attach a time-stamp to each architected register. For example, when simulating an instruction

“R1 := R20 + R23”,

we set the time-stamp of R1 to one plus the maximum of the time-stamps of R20 and R23. The program counter also has a time-stamp, so when executing

“BRANCH-IF-ZERO R3 +5”,

which branches forward 5 instructions if R3 is zero, we can “max-in” the time-stamp of R3 to the stamp of the program counter. When executing a memory instruction, we can, for example, keep a single time-stamp on the memory system, which effectively serializes all store instructions, and makes every load instruction depend on the most recent store instruction (even if that store was to a different location.) These simulation rules would compute the critical path for a processor with an infinite window, and an infinite number of functional units, no memory parallelism, and no branch speculation.

It turns out that we can modify the time-stamping rules to handle many more interesting variations. We implemented time-stamping rules that model the delays induced by a limited fetch width from an infinite instruction or trace cache [14, 11] with a hybrid branch predictor [7] and misprediction penalties, by a limited window size with three different window refilling policies (wrap-around, compressing, and flushing), by a limited number of specialized, pipelined functional units assigned to oldest requesting instructions, and by an out-of-order load/store unit [4]. By maintaining multiple time-stamps for each state and resource, we can simulate a number of different processors concurrently in one simulation run.

Time-stamping has allowed us to model, with little time overhead, many processor features, but not all. For the simulation results presented here, we assume that all memory references hit in either the I-cache or the D-cache. We are developing a simulation that models cache behavior. It is difficult for us to accurately model a cache using time-stamping, however, because we process instructions in the serial execution order, not in the order in which they are executed by an out-of-order processor. For the same reason, it is difficult for us to accurately model a non-pipelined functional unit, such as a divider.

We have also not found an efficient way to model the effect of a limited number of functional units and a wraparound window in which the next instruction scheduled is the ready instruction in the lowest-numbered window entry, i.e. when the window is wrap-around but the scheduler is not. We believe that several existing processors use such a scheduler. Such a scheduler is likely to be worse than a pure wrap-around scheduler: it is usually preferable to schedule an older instruction over a younger instruction so that it can be retired sooner. One problem with a non-wrap-around scheduler on a wrap-around window is that it can exhibit non-monotonic behavior.<sup>4</sup> It can be very difficult to write good compilers for processors that exhibit non-monotonic behavior.

The earliest time-stamping processor simulator that we know of was implemented for the GITA tagged-token dataflow architecture [8]. We are also aware of a time-stamping simulation developed independently by Intel for the Pentium Pro processor.

<sup>4</sup>A non-monotonicity in a processor is a situation where adding an instruction to the inner loop of a program can speed up the program [5].

Common	Hybrid Branch Predictor (see [7])	
Characteristics:	Predictor selection 4096 2-bit counters (1024KB)	
	Local Predictor 4096 2-bit counters (1024KB)	
	Global Predictor (gshare w/ 3 bits) 8192 2-bit counters (2048KB)	
	32 Entry jump prediction stack	
	32 Entry load/store unit	
	Instruction Latencies:	
	Integer ALU (non-multiplication)	1 (2) Cycles
	Integer Multiply	7
	Branch	1 (2)
	Memory	3
	Floating Point (non-division)	4
	FP Single Precision Division	12
	FP Double Precision Division	15
$\alpha$	32 Entry Window	$\beta$ 128 Entry Window
	4-wide fetch	8-wide fetch
	Fetch until taken branch (except for <code>gccr</code> )	Fetch until mispredicted branch (except for <code>gccr</code> )
	2 integer ALUs	4 integer ALUs
	2 branch units	4 branch units
	2 memory ports	4 memory ports
	1 integer multiplier	4 integer multiplier
	2 floating point units	4 floating point units

Figure 9: Processor Characteristics

## The Simulated Processors

We simulated two different processors using our time-stamping simulator. Both processors implement the 21264’s instruction set. The first processor, called  $\alpha$ , resembles today’s commercial processors and provides a baseline for our study. The  $\alpha$  has a 4-instruction fetch width and a single 32-instruction reordering buffer shared by all instructions. The processor fetches an unaligned block of four statically adjacent instructions at a time. The  $\alpha$  can issue up to nine instructions at a time. The functional units, their numbers, and their latencies are described in Figure 9.

The second processor that we simulated, called  $\beta$ , approximates a processor that we believe could be built using our redesigned superscalar components and other recent advances. The  $\beta$  processor wakes up, schedules, and issues instructions from a single 128-instructions reordering buffer. Using a trace cache[14, 11], the processor fetches an unaligned *dynamic* sequence of eight instructions at a time. Fetching of instructions only incurs delays when a mispredicted branch is encountered, rather than on every branch. The  $\beta$  can issue up to twenty instructions at a time. The functional units, their numbers, and their latencies are also described in Figure 9.

The two processors,  $\alpha$  and  $\beta$ , share a number of characteristics. Both processors use a hybrid branch predictor that dynamically chooses between two branch predictors and incurs a 3-cycle penalty on a branch misprediction [7]. The branch predictor tables are the same size for both processors. (It may be that a larger branch predictor would make sense for a larger processor.) Both also have fully pipelined functional units and infinite size caches.

The two processors differ in two important aspects: the structure of their reordering buffer and the use of bypasses. The  $\alpha$  processor uses a compressing reordering buffer much like the 21264, while the  $\beta$  uses a wrap-around reordering buffer. A compressing reordering buffer can make better use of its entries. On every clock cycle, the  $\alpha$ ’s compressing buffer retires all instructions that have finished executing and compresses all remaining entries by pushing them up to the top of the buffer. Thus, on every clock cycle, all unused entries are ready to be refilled with new instructions. In contrast, a wrap-around buffer cannot refill an unused entry until all older instructions have finished. All the circuits described in this paper can operate on a compressing buffer just as well as a wrap-around one<sup>5</sup>; however, we do not know how to compress a 128-instruction reordering buffer quickly. For this reason, the  $\beta$  assumes only a wrap-around buffer.

Unlike the  $\alpha$ , the  $\beta$  also suffers from lack of bypassing. In today’s processors, bypass paths allow many dependent

<sup>5</sup>In fact, the schedule and commit logic can be made to run faster on a compressing buffer by eliminating segment bits.

		Minimum Latency = 1			Minimum Latency = 2			EV6/700
		$\alpha$						
(4-fetch)		Wrap	Compress	Flush	Wrap	Compress	Flush	IPC
(int)	go	2.20	<b>2.22</b>	1.89	1.96	2.05	1.59	1.03
	gcc	2.40	<b>2.45</b>	2.09	2.13	2.24	1.75	2.13
	compress	1.64	<b>1.68</b>	1.33	1.54	1.68	1.12	1.37
	li	2.62	<b>2.67</b>	2.21	2.44	2.49	1.97	1.27
	jpeg	2.63	<b>2.70</b>	2.30	2.39	2.64	1.79	2.75
	perl	2.54	<b>2.86</b>	2.00	2.18	2.52	1.71	1.11
	vortex	3.11	<b>3.14</b>	2.53	2.87	3.11	2.25	1.99
(fp)	tomcatv	2.50	<b>2.53</b>	2.21	2.09	2.16	1.76	1.29
	swim	3.52	<b>3.97</b>	2.37	3.52	3.97	2.37	0.99
	su2cor	2.45	<b>2.53</b>	2.14	2.01	2.13	1.70	0.80
	hydro2d	2.59	<b>3.43</b>	1.72	2.58	3.43	1.71	1.38
	mgrid	3.34	<b>3.55</b>	2.18	3.33	3.54	2.17	2.10
	applu	2.52	<b>3.20</b>	1.71	2.49	3.17	1.68	0.86
	turb3d	3.46	<b>3.54</b>	2.64	3.42	3.53	2.54	1.78
	apsi	2.46	<b>3.12</b>	1.76	2.44	3.10	1.74	1.23
	fpppp	2.56	<b>3.12</b>	1.76	2.55	3.10	1.74	2.04
	wave5	2.68	<b>3.42</b>	1.91	2.59	3.32	1.82	1.11

All benchmarks warmed up for 512M ( $2^{29}$ ) instructions, and then measured over the next 512M instructions.

Figure 10: Simulated IPC for the  $\alpha$  processor and its variations.

instructions to issue back to back. With twenty functional units, we assume that the  $\alpha$  will require an additional clock cycle between certain types of dependent instructions. We assume that instructions with multiple-cycle execution latencies can overlap their execution with the precharging of their results' paths, allowing dependent instructions to issue without delay. Instructions with one-cycle execution latencies cannot precharge their results' paths, however, because their dependents have not yet been scheduled. As a result, one-cycle instructions effectively cost a two-cycle delay.

### Our Simulation Results

We ran a number of simulations in order to better understand the performance of the  $\alpha$  and the  $\beta$  and the performance loss associated with the use of a wrap-around buffer and the lack of bypassing.

Figures 10 and 11 shows the results of our simulations. The figure shows the average instruction-level parallelism of the SPEC cpu95 benchmarks [15] as measured by our time-stamping simulator. The benchmarks were compiled by and for an Alpha 21164 processor using the Digital C compiler invoked with

```
cc -migrate -std1 -O5 -ifo -om,
```

which is the standard “vendor-supplied” compiler option for compiling SPEC95. Since the 21164 is an in-order processor, our code was not optimized for the out-of-order features of the 21264. The highlighted columns in the figures correspond to the two described processors,  $\alpha$  and  $\beta$ . Despite its limitations, the 128-buffer  $\beta$  substantially outperforms the 32-buffer  $\alpha$ . Even on benchmarks such as gcc which have relatively little parallelism, the wide-window processor shows a significant performance gain.

The remaining columns vary the structure of the reordering buffer and the bypassing policy. The columns labeled “Wrap” and “Compress” report the performance of a wrap-around reordering buffer and a compressing reordering buffer, respectively. The columns labeled “Flush” are a strawman architecture in which when the window fills up, all instructions must retire before the next instruction can run. We found that the compressing window only buys a small amount of additional performance, and the flushing window is much slower.

The columns labeled “Minimum Latency = 2” force all instructions to have at least a latency of two clock cycle in order to model the lack of bypassing in the  $\beta$  processor. Increasing the latency of the integer unit to two incurs a fairly significant overhead, but does not outweigh the benefit of a large reordering buffer.

		Minimum Latency = 1			Minimum Latency = 2			EV6/700
(8-fetch)		Wrap	Compress	Flush	$\beta$			
		Wrap	Compress	Flush	Wrap	Compress	Flush	
(int)	go	3.11	3.11	2.79	<b>2.72</b>	2.73	2.36	—
	gcc	3.32	3.33	3.05	<b>2.85</b>	2.87	2.55	—
	compress	2.01	2.01	1.84	<b>1.99</b>	1.99	1.68	—
	li	3.33	3.33	3.11	<b>3.04</b>	3.04	2.82	—
	jpeg	4.82	4.91	4.27	<b>4.50</b>	4.64	3.46	—
	perl	3.23	3.23	2.92	<b>2.75</b>	2.75	2.46	—
	vortex	4.61	4.61	4.12	<b>4.31</b>	4.33	3.73	—
(fp)	tomcatv	3.24	3.24	3.05	<b>2.59</b>	2.59	2.38	—
	swim	6.31	6.36	4.16	<b>6.30</b>	6.36	4.15	—
	su2cor	3.10	3.10	2.91	<b>2.46</b>	2.46	2.25	—
	hydro2d	5.34	5.56	3.73	<b>5.22</b>	5.42	3.66	—
	mgrid	5.88	5.89	4.20	<b>5.84</b>	5.87	4.16	—
	applu	4.67	5.03	3.24	<b>4.59</b>	4.99	3.13	—
	turb3d	6.65	6.70	5.19	<b>6.35</b>	6.43	4.86	—
	apsi	4.66	4.77	3.34	<b>4.56</b>	4.70	3.25	—
	fpppp	3.75	3.83	2.85	<b>3.73</b>	3.81	2.83	—
	wave5	4.96	5.14	3.52	<b>4.73</b>	4.89	3.36	—

All benchmarks warmed up for 512M ( $2^{29}$ ) instructions, and then measured over the next 512M instructions.

Figure 11: Simulated IPC for the  $\beta$  processor and its variations.

For comparison, we also report in the right-most column the average instruction-level parallelism achieved by a real machine, an Alpha GS140 21264 700MHz EV6 (the rightmost column.) We have derived these numbers by measuring the number of instructions executed for each SPEC benchmark and dividing by the clock speed and by the published runtimes. This IPC is only an estimate, since the published CPU95 results for the 21264 are compiled with a newer compiler, and so the instruction counts are different. This IPC should resemble the IPC of our  $\alpha$  processor.

Comparing the  $\alpha$  to the EV6, we conclude that our simulation can reasonably model the performance of some benchmarks (gcc, compress, i jpeg), but predicts a much higher IPC for other benchmarks (go, swim, su2cor) than is actually achieved by a 21264. We expect that many variables contribute to the inaccuracy. First, the two processors differ in a number of significant ways. The 21264 has an integer window of 20 instructions and an FP window of 15 instructions, with a clustering mechanism that can further increase latencies. The  $\alpha$ , on the other hand, has one window of 32 instructions. For example, [14] showed that the performance of the go program is very sensitive to window size even when holding everything else constant, and thus our 32-entry window may be better than the 21264’s split 40-entry window. The IPC numbers in [14] match ours fairly closely for go. The  $\alpha$  also has two general FPU’s instead of one FP adder and one FP multiplier, two general-purpose memory ports instead of one for each window, and a pipelined divider. In addition, the  $\alpha$  fails to model cache misses including I/O traffic.

To isolate the impact of the trace cache on the performance, we also ran the gcc SPEC benchmark on the small processor with a trace cache, and on the big processor with the non-trace I-cache. Figure 12 shows the impact of the trace cache. Surprisingly, the trace cache is responsible for a relatively small part of the speedup.

## 6 Conclusion

Our studies were done with a  $0.25\mu\text{m}$  aluminum process that is already obsolete. We scaled our layout into UMC’s  $0.18\mu\text{m}$  copper/low-K technology, without reoptimizing it for the process, and the SPICE time is about 0.6ns for the scheduler. We conclude that processors with 128-wide windows and many functional units may soon be practical.

Architectural ideas such as clustering [9], SMT [16], and trace caches [14, 11] are largely orthogonal to our results. For example, processors should still be clustered, but the size of the windows can be made much larger.

Patt et al [12] have been arguing for several years that the best use of a chip with a billion transistors is to build a single large uniprocessor. Our results suggest that such a processor can and should be built.

$\alpha$	Minimum Latency = 1			Minimum Latency = 2			GS140, 6/700 Spec Base
	Wrap	Compress	Flush	Wrap	Compress	Flush	
<code>gcc<sub>I</sub></code>	2.40	2.45	2.09	2.13	2.24	1.75	2.13
<code>gcc<sub>T</sub></code>	2.49	2.58	2.11	2.18	2.34	1.76	—
$\beta$							
<code>gcc<sub>I</sub></code>	3.14	3.14	2.95	2.74	2.74	2.49	—
<code>gcc<sub>T</sub></code>	3.32	3.33	3.05	2.85	2.87	2.55	—

All benchmarks warmed up for 512M ( $2^{29}$ ) instructions, and then measured over the next 512M instructions.

Figure 12: The impact of the trace cache. The runs labelled `gccI` used a traditional I-cache, where the runs labelled `gccT` used a trace cache model.

## References

- [1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1990.
- [2] James A. Farrell and Timothy C. Fischer. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.
- [3] Dana S. Henry, Bradley C. Kuszmaul, and Vinod Viswanath. The Ultrascalar processor—an asymptotically scalable superscalar microarchitecture. In *The Twentieth Anniversary Conference on Advanced Research in VLSI (ARVLSI'99)*, pages 256–273, Atlanta, GA, 21–24 March 1999. <http://ee.yale.edu/papers/usmemo3.ps.gz>.
- [4] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.
- [5] Nathaniel A. Kushman. Performance nonmonotonicities: A case study of the UltraSPARC processor. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1998. <ftp://theory.lcs.mit.edu/pub/cilk/kushman-ms-thesis.ps.gz>.
- [6] Bradley C. Kuszmaul, Dana S. Henry, and Gabriel H. Loh. A comparison of scalable superscalar processors. In *The Eleventh ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*, pages 126–137, St. Malo, France, 27–30 June 1999. An early version is available as Ultrascalar Memo 4, 27 January 1999, from the Yale University Computer Architecture and Engineering Group, 51 Prospect Street, New Haven, CT 06525 <http://ee.yale.edu/papers/usmemo4.ps.gz>.
- [7] Scott McFarling. Combining branch predictors. Technical Note TN-36, Digital Western Research Laboratory, 250 University Avenue, Palo Alto, CA 94301, June 1993.
- [8] Rishiyur S. Nikhil, P. R. Fenstermacher, and J. E. Hicks. Id world reference manual (for LISP machines). Unnumbered technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, Computations Structures Group, 1988. Supersedes Dinarte R. Morais, Id World: User's Manual, Computations Structures Group Memo 266, June 1986.
- [9] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, pages 206–218, Denver, Colorado, 2–4 June 1997. ACM SIGARCH and IEEE Computer Society TCCA. <http://www.ece.wisc.edu/~jes/papers/isca.ss.ps>. See also [10].
- [10] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-96-1328, University of Wisconsin, Madison, 19 November 1996. <ftp://ftp.cs.wisc.edu/sohi/complexity.report.ps.Z>.
- [11] Sanjay Jeram Patel, Daniel Holmes Friendly, and Yale N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, Computer Science and Engineering, University of Michigan, 7 May 1997. [http://www.eecs.umich.edu/HPS/hps\\_tracecache.html](http://www.eecs.umich.edu/HPS/hps_tracecache.html).

- [12] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, September 1997. <http://www.computer.org/computer/co1997/r9051abs.htm>.
- [13] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.
- [14] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO 30)*, pages 138–148, Research Triangle Park, North Carolina, 1-3 December 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.
- [15] SPEC (Standard Performance Evaluation Corporation). SPEC 95 CPU performance benchmarks. 10754 Ambassador Drive, Suite 201, Manassas, VA 20109, 1995. <http://www.specbench.org/>.
- [16] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 392–403, Santa Margherita Ligure, Italy, 22–24 June 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News*, 23(2), May 1994.
- [17] Ben Verghese. SimOS-Alpha. <http://www.research.digital.com/wrl/projects/SimOS/>, February 1998.
- [18] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. VLSI Systems Series. Addison-Wesley, 1985.