A Comparison of Scalable Superscalar Processors

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

Abstract

The poor scalability of existing superscalar processors has been of great concern to the computer engineering community. In particular, the critical-path lengths of many components in existing implementations grow as $\Theta(n^2)$ where n is the fetch width, the issue width, or the window size. This paper describes two scalable processor architectures, the Ultrascalar I and the Ultrascalar II, and compares their VLSI complexities (gate delays, wire-length delays, and area.) Both processors are implemented by a large collection of ALUs with controllers (together called *execution stations*) connected together by a network of parallel-prefix tree circuits. A fat-tree network connects an interleaved cache to the execution stations. These networks provide the full functionality of superscalar processors including renaming, out-of-order execution, and speculative execution.

The difference between the processors is in the mechanism used to transmit register values from one execution station to another. Both architectures use a parallel-prefix tree to communicate the register values between the execution stations. The Ultrascalar I transmits an entire copy of the register file to each station, and the station chooses which register values it needs based on the instruction. The Ultrascalar I uses an H-tree layout. The Ultrascalar II uses a mesh-of-trees and carefully sends only the register values that will actually be needed by each subtree to reduce the number of wires required on the chip.

The complexity results are as follows: The complexity is described for a processor which has an instruction-set architecture containing L logical registers and can execute n instructions in parallel. The chip provides enough memory bandwidth to execute up to $M(n)$ memory operations per cycle. (M is assumed to have a certain regularity property.) In all the processors, the VLSI area is the square of the wire delay. The Ultrascalar I has gate delay $O(\log n)$ and wire-delay

$$\tau_{\text{wires}} = \begin{cases} \Theta(\sqrt{n}L) & \text{if } M(n) \text{ is } O(n^{1/2-\epsilon}), \\ \Theta(\sqrt{n}(L+\log n)) & \text{if } M(n) \text{ is } \Theta(n^{1/2}), \text{ and} \\ \Theta(\sqrt{n}L + M(n)) & \text{if } M(n) \text{ is } \Omega(n^{1/2+\epsilon}) \end{cases}$$

for $\epsilon > 0$. The Ultrascalar II has gate delay $\Theta(\log L + \log n)$. The wire delay is $\Theta(n)$, which is optimal for $n = O(L)$. Thus, the Ultrascalar II dominates the Ultrascalar I for $n = O(L^2)$, otherwise the Ultrascalar I dominates the Ultrascalar II.

We introduce a hybrid ultrascalar that uses a two-level layout scheme: Clusters of execution stations are layed out using the Ultrascalar II mesh-of-trees layout, and then the clusters are connected together using the H-tree layout of the Ultrascalar I. For the hybrid (in which $n \geq L$) the wire delay is $\Theta(\sqrt{nL} + M(n))$, which is optimal. For $n \geq L$ the hybrid dominates both the Ultrascalar I and Ultrascalar II.

We also present an empirical comparison of the Ultrascalar I and the hybrid, both layed out using the Magic VLSI editor. For a processor that has 32 32-bit registers and a simple integer ALU, the hybrid requires about 11 times less area.

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

SPAA '99 Saint Malo, France

1 Introduction

Today's superscalar processors must rename registers, bypass registers, checkpoint state so that they can recover from speculative execution, check for dependencies, allocate execution units, and access multi-ported register files. The circuits employed are complex, irregular, and difficult to implement. Furthermore, the delays through many of today's circuits grow quadratically with issue width (the maximum number of simultaneously fetched or issued instructions), and with window size (the maximum number of instructions within the processor core). It seems likely that some of those circuits can be redesigned to have at most linear delays, but all the published circuits are at least quadratic delay [12, 3, 4]. Chips continue to scale in density. For example, Texas Instruments announced recently a 0.07 micrometer process with plans to produce processor chips in volume production in 2001 [18]. With billion transistor chips on the horizon, this scalability barrier appears to be one of the most serious obstacles for high-performance uniprocessors in the next decade.

This paper compares three different microarchitectures that improve the scalability bounds while extracting the same instruction-level parallelism as today's superscalars. All three processors rename registers, issue instructions out of order, speculate on branches, and effortlessly recover from branch mispredictions. The three processors all implement identical instruction sets, with identical scheduling policies. The only differences between the processors are in their VLSI complexities, which include gate delays, wire delays, and area, and which have implications therefore on clock speeds. The regular structure of the processors not only makes it possible to theoretically analyze the VLSI complexity of the processors, but made it possible for us to quickly implement VLSI layouts to facilitate an empirical comparison.

We analyze the complexity of each processor as a function of three important parameters.

1. L is the *number of logical registers,* which is defined by the instruction-set architecture. L is the number of registers that

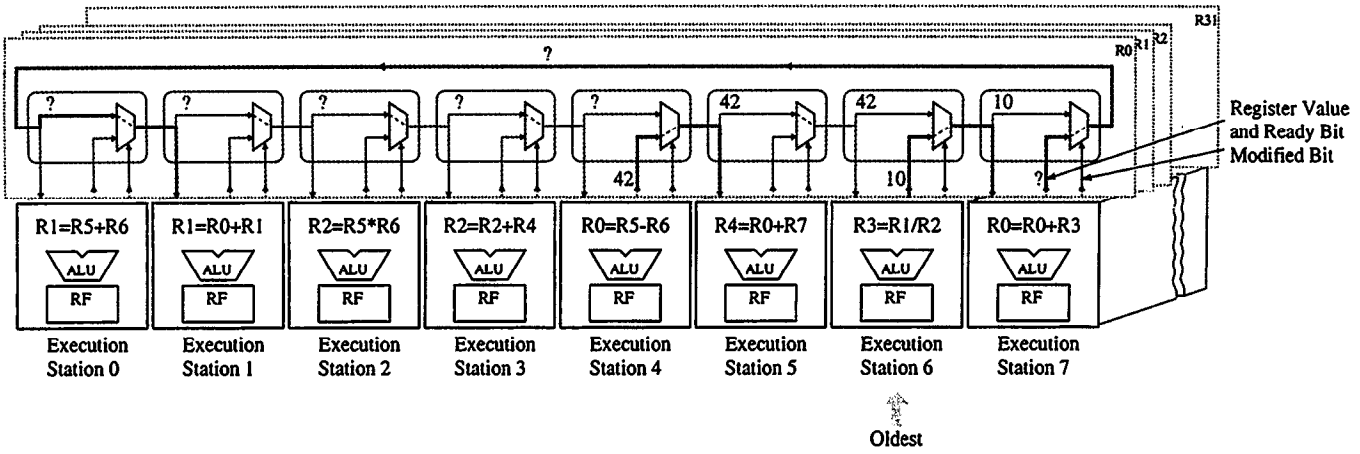

Figure 1: A linear-gate-delay datapath for the Ultrascalar I processor.

the programmer sees, which in most modern processors is substantially less than the number of real registers that the processor implementor employs. A superscalar processor often uses more real registers than logical registers to exploit more instruction-level parallelism than it could otherwise.

2. n is the *issue width* of the processor. The issue width determines how many instructions can be executed per clock cycle. This paper does not study the effect of increasing the instruction-fetch width or the window size independently of the issue width. A simplified explanation of these terms follows: The instruction-fetch width is the number of instructions that can enter the processor pipeline on every clock cycle, whereas the issue-width is the number that can leave the processor pipeline on every cycle. It is reasonable to assume that the issue width and the instruction-fetch width scale together. The window size is the number of instructions that can be "in flight" between the instruction fetcher and the issue logic. That is, the window size indicates how many instructions have entered the processor and are being considered for execution. In most modern processors the window size is an order of magnitude larger than the issue width. We do not here consider the window-size independently because it makes no difference to the theoretical results. From an empirical point of view, it is doubtless worth investigating the impact of changing the window size independently from the issue width. We know how to separate the two parameters by issuing instructions to a smaller pool of shared ALUs. Our ALU scheduling circuitry is described elsewhere [6] and fits within the bounds described here.

3. M is the bandwidth provided to memory. The idea is that some programs need quite a bit of memory bandwidth per instruction, whereas others need less. A processor may be optimized for one case over another. The memory bandwidth is naturally viewed as a function of n, since a small processor that can only execute 1 instruction per cycle may need to provide facilities for one memory operation per cycle to achieve high performance, whereas a larger processor that can execute 100 instructions per cycle may only need 10 memory operations per cycle to keep the same program running quickly. The larger processor may need less bandwidth per instruction because it has a better chance to smooth out the memory accesses over time, and because it may be able to use caching techniques to move data directly from one instruction to another without using up memory bandwidth. Thus, we express the memory bandwidth, $M(n)$, as a function of n. We assume that $M(n)$ is $O(n)$, since it makes no sense to provide more memory bandwidth than the total instruction issue rate.

This paper contributes two new architectures, the Ultrascalar II and a hybrid Ultrascalar, and compares them to each other and to a third, the Ultrascalar I. We described the Ultrascalar I architecture in [7], but did not analyze its complexity in terms of L. For L equal to 64 64-bit values, as is found in today's architectures, the improvement in layout area is dramatic over the Ultrascalar I.

This paper does not evaluate the benefits of larger issue widths, window sizes, or of providing more logical registers. Some work has been done showing the advantages of large-issue-width, large-window-size, and large-register-file processors. Lam and Wilson suggest that ILP of ten to twenty is available with an infinite instruction window and good branch prediction [8]. Patel, Evers and Patt demonstrate significant parallelism for a 16-wide machine given a good trace cache [13]. Patt et al argue that a window size of 1000's is the best way to use large chips [14]. Steenkiste and Hennessy [17] conclude that certain compiler optimizations can significantly improve a program's performance if many logical registers are available. And, although the past cannot guarantee the future, the number of logical registers has been steadily increasing over time. The amount of parallelism available in a thousand-wide instruction window with realistic branch prediction and 128 logical registers is not well understood and is outside the scope of this paper. Thus, we are interested in processors that scale well with the issue width, the window size, the fetch width, and with an increasing number of logical registers in the instruction-set architecture.

The rest of this paper describes and compares our three scalable processor designs. Section 2 functionally describes the Ultrascalar I, and analyzes its gate-level complexity. Section 3 provides the VLSI floorplan for the Ultrascalar I and analyzes its area and wire-length complexity. Section 4 describes the Ultrascalar II at gate-level, whereas Section 5 addresses VLSI layout, area, and wire-lengths. Section 6 shows how to build a hybrid of the Ultrascalar I and Ultrascalar II, and analyzes its complexity. Section 7 compares the three processors both analytically and empirically by comparing our VLSI layouts.

2 Ultrascalar I Design

The Ultrascalar I processor achieves scalability with a completely different microarchitecture than is used by traditional superscalar processors. Instead of renaming registers and then broadcasting renamed results to all outstanding instructions, as today's superscalars do, the Ultrascalar I passes the entire logical register file, annotated with ready bits, to every outstanding instruction. Later in this section we shall show how to implement the Ultrascalar I with only logarithmic gate delays, but for ease of understanding, we start with an explanation of circuits that have linear gate-delay. The description given here stands on its own, but a more in-depth description of the Ultrascalar I can be found in [7].

Figure 1 shows the datapath of an Ultrascalar I processor implemented in linear gate-delay. Eight outstanding instructions are shown, analogous to an eight-instruction window in today's superscalars. Each of the eight instructions occupies one *execution station*. An execution station is responsible for decoding and executing an instruction given the data in its register file. At any given time, one of the stations is marked as being the "oldest", with younger stations found to the right of the oldest, and then still younger stations found by wrapping around and progressing left-to-right from Station 0. In Figure 1, Station 6 is the oldest. Thus the complete sequence of instructions currently in the datapath is:

Instruction Sequence	Execution Station
$R_3 = R_1 / R_2$	(6)
$R_0 = R_0 + R_3$	(7)
$R_1 = R_5 + R_6$	(0)
$R_1 = R_0 + R_1$	(1)
$R_2 = R_5 * R_6$	(2)
$R_2 = R_2 + R_4$	(3)
$R_0 = R_5 - R_6$	(4)
$R_4 = R_0 + R_7$	(5)

The execution stations communicate with each other using rings of multiplexers as illustrated in Figure 1. There are L rings of multiplexers, one for each logical register defined by the instruction set. Each multiplexer ring carries the latest value of its logical register and a ready bit to successive execution stations. Any instruction that modifies the ring's register inserts a new value and a new ready bit into the ring. The ready bit indicates whether or not the instruction has already computed the register's value. The oldest station inserts the initial value of every register into each ring.

The execution stations in Figure 1 continually refill with new instructions. Stations holding finished instructions are reused as soon as all earlier instructions finish. The station holding the oldest unfinished instruction becomes the oldest station on the following clock cycle.

Figure 2 illustrates the internal structure of each execution station. Each station includes its own functional units (ALU), its own register file, instruction decode logic, and control logic which is not shown. The register file holds the values of all the registers and their ready bits as written by preceding instructions. The register file is updated at the beginning of every clock cycle. Each station, other than the oldest, latches all of its incoming values and ready bits into its register file. Next, each station reads its arguments, each of which is a value and a ready bit, from the register file. Based on the arguments, the ALU computes the result's value and ready bit. Then the station inserts the result into the outgoing register datapath. The rest of the outgoing registers are set from the register file. The decode logic generates a *modified* bit for every logical register, indicating wheather the station has modified the register's value and ready bit. The modified bit is used to control the register's multiplexer in the datapath of Figure 1. Typically, only the result register

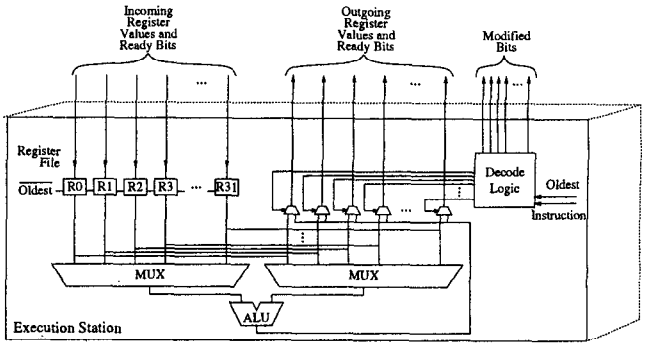

Figure 2: An Ultrascalar I execution station.

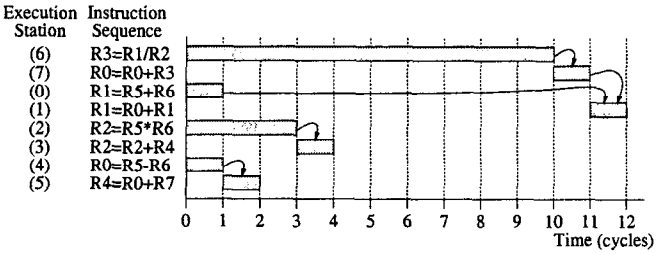

Figure 3: Timing diagram showing the relative time during which each instruction in the sequence executes. Arrows indicate true data dependencies.

is marked modified. But in the case of the oldest station, all registers are marked modified. Finally at the end of each clock cycle, newly computed results propagate through the datapath of Figure 1.

Figure 1 shows a snapshot of some of the values passing through the Ultrascalar I datapath. The multiplexer ring at the forefront of the figure carries the latest values of register R_0 and their ready bits. The flow of data through the ring, from writers of R_0 to readers of R_0, is marked in bold. The current setting of every multiplexer is indicated by dashed lines. The oldest station, Station 6, has set its multiplexer select line high, thus inserting the initial value of R_0 into the ring. The initial value, equal to 10, is marked ready. We will indicate values that are not ready with a question mark. The instructions in Stations 7 and 4 modify R_0. At the time of the snapshot, the instruction in Station 7 has not yet computed its result. It has inserted a low ready bit into the chain instead, informing the instructions in Stations 0–4 that the value of R_0 is not yet ready. The instruction in Station 4, on the other hand, has already computed its result. It has set the value of R_0 to 42 and informed Stations 5 and 6 that R_0 is ready. Recall that Station 6 ignores the incoming information: Since it is the oldest station, it does not latch incoming values into its register file.

Figure 3 illustrates the dynamic behavior of the Ultrascalar datapath. The figure shows the relative time at which each instruction in our eight-instruction sequence computes. We assume that division takes 10 clock cycles, multiplication 3, and addition 1. Note that the datapath in Figure 1 exploits the same instruction-level parallelism as today's superscalars. Instructions compute out of order using renamed register values. The instruction in Station 4, for example, computes right away since it is not dependent on any of the earlier outstanding instructions. In contrast, the earlier instruction in Station 7, which also modifies R_0, does not issue for ten more clock cycles. This timing diagram is exactly what would

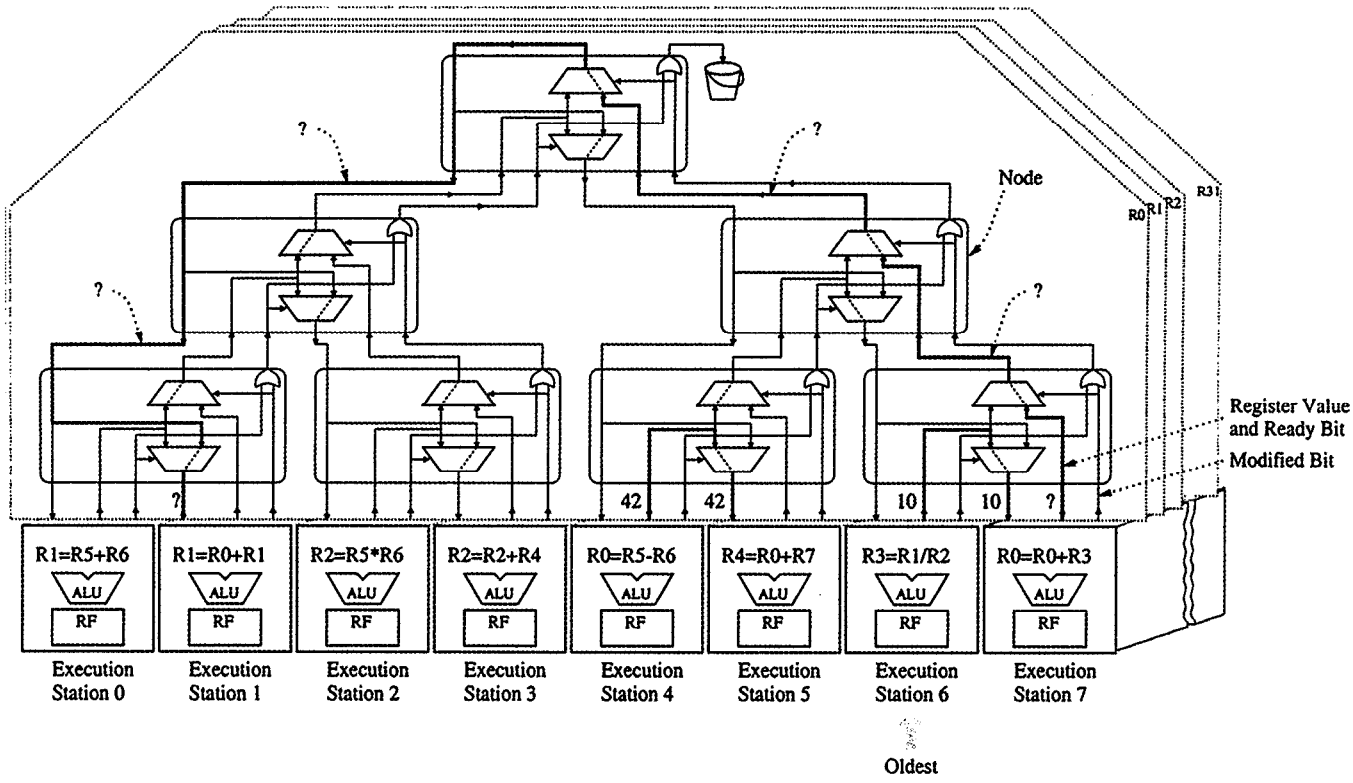

Figure 4: A logarithmic-gate-delay datapath for the Ultrascalar I processor.

be produced in a traditional superscalar processor that has enough functional units to exploit the parallelism of the code sequence.

The Ultrascalar I datapath described in Figure 1 does not scale well. The datapath can propagate a newly computed value through the entire ring of multiplexers in a single clock cycle. Each multiplexer has a constant gate delay and the number of multiplexers in the ring is equal to the number of outstanding instructions, n. As a result, the processor's clock cycle is $O(n)$ gate delays.

One way to reduce the processor's asymptotic growth in clock cycle is by replacing each ring of multiplexers with a cyclic, segmented, parallel-prefix (CSPP) circuit [5]. Figure 4 shows the Ultrascalar I datapath redesigned to use one CSPP circuit for each logical register. Each parallel prefix circuit has exactly the same functionality and the same interface as the multiplexer ring that it has replaced. As in the linear implementation, the cyclic, segmented parallel prefix circuit at the forefront of Figure 1 carries the latest values of register R_0 and their ready bits. The flow of data through the prefix, from each writer of R_0 to each reader of R_0, is again shown in bold, and multiplexer settings are indicated by dashed lines. We see the three current settings of R_0 (10,?, and 42) propagating through the tree, reaching the same stations that they reached in Figure 4.

The CSPP circuit can be understood as follows. A (noncyclic) segmented parallel prefix circuit computes, for each node of the tree, the accumulative result of applying an associative operator to all the preceding nodes up to and including the nearest node whose segment bit is high. (See [1] for a discussion of parallel prefix and in particular see Exercise 29.2-8 for segmented parallel prefix.) In this instance, the associative operator ($a \otimes b = a$) simply passes earlier values and the segment bit identifies instructions that insert new register values into the parallel prefix. The resulting segmented parallel prefix circuit computes for each station the most recently modified value of the register. In addition, we have turned each segmented parallel prefix circuit into a *cyclic*, segmented parallel prefix (CSPP) circuit by tying together the data lines at the top of the tree and discarding the top segment bit. In our CSPP circuit the preceding stations wrap around until a station is found that has modified the register. With CSPP circuits implementing the datapath, the circuit has gate delay $O(\log n)$.

The Ultrascalar I uses several more CSPP circuits in order to correctly sequence instructions. All of the remaining CSPP circuits have the same structure. They all use the 1-bit-wide associative operator $a \otimes b = a \wedge b$. Figure 5 illustrates. By having the oldest station raise its segment bit, this CSPP circuit can compute for each station whether all the earlier stations have met a particular condition. In Figure 5, Station 6 is the oldest and has raised its segment bit. Stations 6,7,0,1, and 3 have met the condition and have raised their inputs to the CSPP. The circuit outputs a high to Stations 7,0,1 and 2 informing them that all earlier stations have met the condition.

One instance of the circuit in Figure 5 is used to compute which station is the oldest on every clock cycle. The circuit informs each station whether all preceding stations have finished their instructions. If a station has not yet finished executing and all preceding stations have, it becomes the oldest station on the next clock cycle. If a station has finished executing and so have all the preceding stations, the station becomes deallocated and can be refilled with a new instruction on the next clock cycle.

Two more instances of the circuit in Figure 5 serialize memory accesses. The first circuit informs each station whether all preceding stations have finished storing to memory. The second whether all preceding loads have finished. A station cannot load from mem-

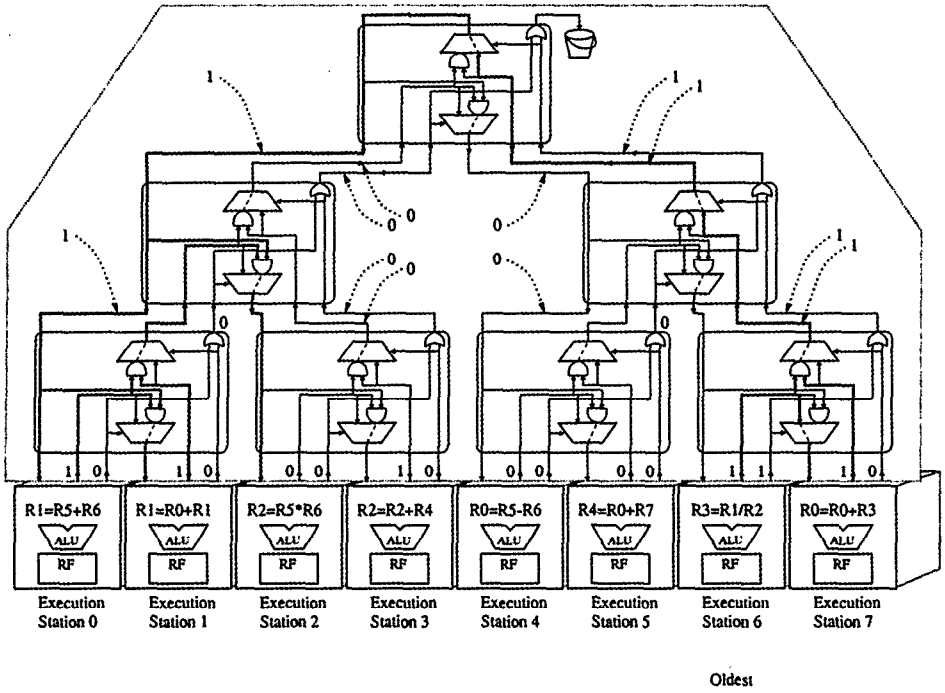

Figure 5: A 1-bit-wide cyclic, segmented parallel prefix circuit (CSPP) using the associative operator $a \otimes b = a \wedge b$.

ory until all preceding stores have finished. A station cannot store to memory until all preceding loads and stores have finished.

A third instance of the circuit in Figure 5 helps support branch speculation. The circuit informs each station whether all preceding stations have confirmed their speculative branches and committed to executing. A station cannot modify memory, for example, until all preceding stations have committed.

In addition to connecting to the above CSPP circuits, each station in the Ultrascalar I must somehow connect to instruction memory and data memory. There are many ways to implement such a connection. For one, the Ultrascalar I can use the same interfaces as today's superscalars. To the memory subsystem, each station looks just like an entry in a wrap-around instruction window of one of today's processors. Of course, in order to take advantage of our scalable processor datapath, the memory interface also should scale. We propose to connect the Ultrascalar I datapath to an interleaved data cache [7] and to an instruction trace cache [20, 15] via two fat-tree or butterfly networks [10]. This allows one to choose how much bandwidth to implement by adjusting the fatness of the trees.

The internal structure of the Ultrascalar datapath provides a mechanism for fast and graceful recovery from the speculative execution of a mispredicted branch. The proper state of the register file for execution stations containing mispredicted instructions is provided by previous execution stations and the committed register file. Nothing needs to be done to recover from misprediction except to fetch new instructions from the correct program path. This is true for all the Ultrascalar models presented in this paper.

3 Ultrascalar I Floorplan and Analysis

While the previous section has described the Ultrascalar I datapath and analyzed its asymptotic gate delay, it has not addressed the datapath's wire delay or area. This is because, unlike gate delay, wire delay and area depend on a particular VLSI layout. In this section, we describe a 2-dimensional VLSI layout for the Ultrascalar I and analyze its asymptotic wire delay and area.

Figure 6 shows the floorplan of a 16-station Ultrascalar I datapath. The 16 execution stations are arranged in a two-dimensional matrix and connected to each other and to memory exclusively via networks layed out with H-tree layouts. The parallel prefix trees of Figure 4 and Figure 5 form H-trees of constant bandwidth along each link. Fat-trees that interface stations to memory form fat H-trees with bandwidth increasing along each link on the way to the root. Each node P in Figure 6 consists of one parallel prefix node from Figure 4 and propagates the value of one logical register. Each node M consists of one fat-tree node and routes a number of memory accesses. (The three 1-bit-wide parallel prefix trees that sequence instructions (Figure 5) are not shown since their area is only a small constant factor, typically a hundredth or less, of the the area taken up by the register-propagating parallel prefixes.)

To determine the area and wire delays, we start by determining the side length of an n-station Ultrascalar I layout, $X(n)$. The sidelength, $X(n)$, is equal to twice the sidelength of an $n/4$-wide Ultrascalar, $X(n/4)$, plus the width of the wires and nodes connecting the four $n/4$-wide Ultrascalars. Connecting the registers, there are $\Theta(L)$ wires and $\Theta(L)$ prefix nodes of constant side-length. In addition, there are $\Theta(M(n))$ wires and one fat tree node used to provide memory bandwidth out of a subtree of n execution stations. The fat tree node's sidelength is proportional to its bandwidth and therefore $\Theta(M(n))$. In the base case, a 1-station-wide Ultrascalar has width $\Theta(L)$. Thus we have the following recurrence:

$$X(n) = \begin{cases} \Theta(L) + \Theta(M(n)) + 2X(n/4) & \text{if } n > 1, \\ \Theta(L) & \text{otherwise.} \end{cases}$$

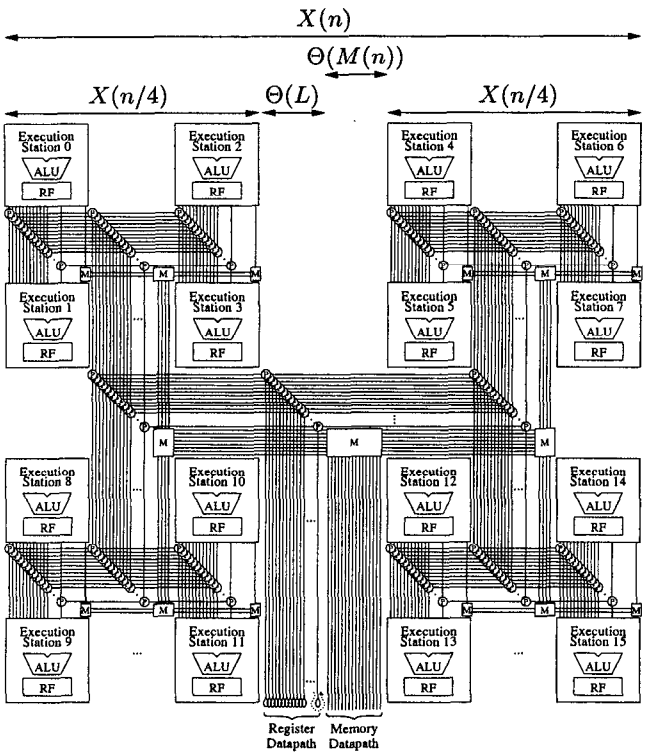

Figure 6: A floorplan of a 16-instruction Ultrascalar I with full memory bandwidth.

This recurrence has solution

$$X(n) = \begin{cases} \Theta(\sqrt{n}L) & \text{if } M(n) = O(n^{1/2-\epsilon}) \text{ for } \epsilon > 0, \quad \text{[Case 1]} \\ \Theta(\sqrt{n}(L + \log n)) & \text{if } M(n) = \Theta(n^{1/2}), \quad \text{[Case 2]} \\ \Theta(\sqrt{n}L + M(n)) & \text{if } M(n) = \Omega(n^{1/2+\epsilon}) \text{ for } \epsilon > 0. \quad \text{[Case 3]} \end{cases}$$

(We assume for Case 3 that M meets a certain "regularity" requirement, namely that $M(n/4) \leq cM(n)/2$ for some c and for all sufficiently large n. See [1] for techniques to solve these recurence relations and for a full discussion of the requirements on M.)

The chip area is simply the square of $X(n)$. If L is viewed as a constant, then in two-dimensional VLSI technology the bounds for Cases 1 and 3 are optimal; and the bound for Case 2 is optimal to within a factor of $\log n$ (i.e., it is *near optimal.*) Case 1 is optimal because to lay out n executions stations will require a chip that is $\Omega(\sqrt{n})$ on a side. Case 2 is similar to Case 1. Case 3 is optimal because to provide external memory bandwidth of $M(n)$ requires a side length of $\Omega(M(n))$.

Next we compute the wire-delays (or speed-of-light delays) of the Ultrascalar I. Wire delay can be made linear in wire length by inserting repeater buffers at appropriate intervals [2]. Thus we use the terms wire delay and wire length interchangeably here.

Given the size of the bounding box for an Ultrascalar I, we can compute the longest wire distances and consequently the longest wire delays as follows. We observe that the total length of the wires from the root to an execution station is independent of which execution station we consider. Let $W(n)$ be the wire length from the root to a leaf of an n-wide Ultrascalar I. We observe that every datapath signal goes up the tree, and then down (it does not go up, then down, then up, then down, for example.) Thus, the longest datapath signal is $2W(n)$.

The wire length, $W(n)$, is the sum of

- the distance from the edge of the Ultrascalar to the central block containing the parallel prefix switches (distance $X(n/4)$), plus
- the distance through the central block (which is $\Theta(L) + \Theta(M(n))$ on a side), plus
- the distance from the root of an $n/2$-wide Ultrascalar to its leaves (distance $W(n/2)$).

Thus we have the following recurrence for $W(n)$:

$$W(n) = \begin{cases} X(n/4) + \Theta(L + M(n)) + W(n/2) & \text{if } n > 1, \\ O(L) & \text{otherwise.} \end{cases}$$

This recurrence has solution

$$W(n) = \Theta(X(n)).$$

That is, the wire lengths are the same as the side lengths of the chip to within a constant factor. Again, if L is viewed as a constant, and if one assumes that on any processor the far corners of the chip must communicate, then this wire length is optimal or near-optimal.

It is better to not view the number of logical registers, L, as a constant, however. Historically, the number of logical registers specified by an instruction set architecture has increased with each new generation of architectures, from instruction sets with a single accumulator register [19] to modern RISC instruction sets with 64 64-bit-wide registers, such as the Digital/Compaq Alpha procesor[16]. It is widely believed that to exploit more parallelism requires more logical registers, as well as more physical registers.

If we do treat L as a constant, then it is a huge constant. Today's processors typically have 64 or more 32-bit or 64-bit registers. Even a small processor with 32 32-bit registers would have 1024 wires in each direction along every edge of the tree. For a 64 64-bit register Ultrascalar I, each node of our H-tree floorplan would require area comparable to the entire area of one of today's processors!

4 Ultrascalar II Design

We have designed the Ultrascalar II which avoids many of the inefficiencies associated with passing all L logical registers to every execution station. The Ultrascalar II passes only the argument and result registers to and from each execution station. This section describes and analyzes the Ultrascalar II processor at gate-level. The next section examines the VLSI layout of the Ultrascalar II. We start out by describing a simplified Ultrascalar II with relatively few instructions that does not wrap around. After describing the basic idea, we show how to use a mesh-of-trees to reduce the gate-delays from linear-time to log-time.

Figure 7 shows the design of a 4-instruction Ultrascalar II datapath. For simplicity, the design assumes an instruction set architecture with only four logical registers. The design also assumes that each instruction in the instruction set reads at most two registers and writes at most one. The datapath consists of a register file that holds the initial value of each register, a sequence of four instructions stored in four stations from left to right, a grid-like network, and a memory switch. The grid-like network routes arguments to each station and computes the final value of each register. On every clock cycle, stations with ready arguments compute and newly computed results propagate through the network. Eventually, all stations finish computing and the final values of all the registers are ready. At that time, the final values (upper right corner) are latched

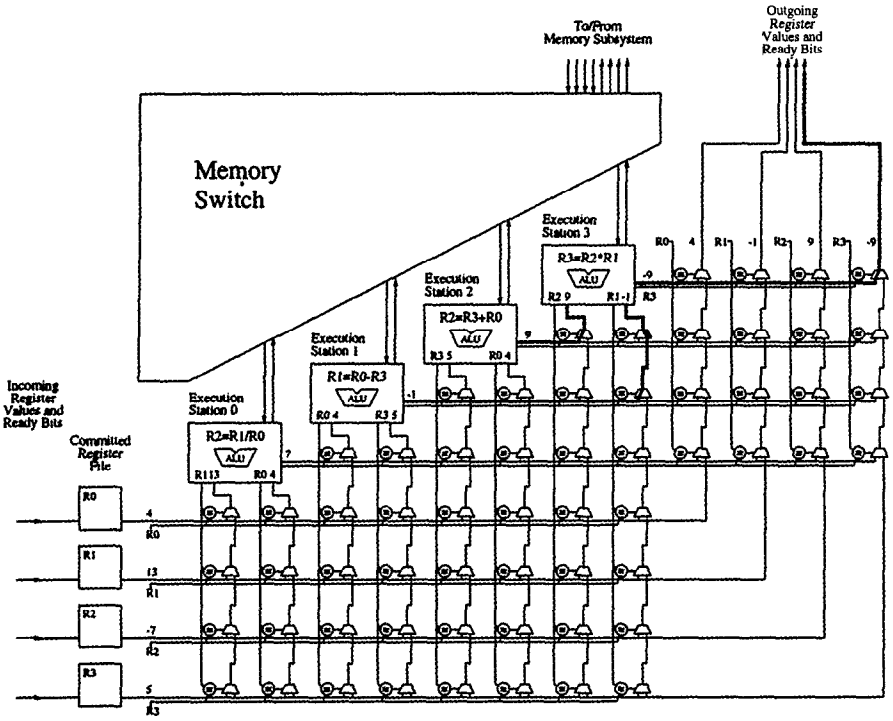

Figure 7: The register datapath and floorplan of the Ultrascalar II.

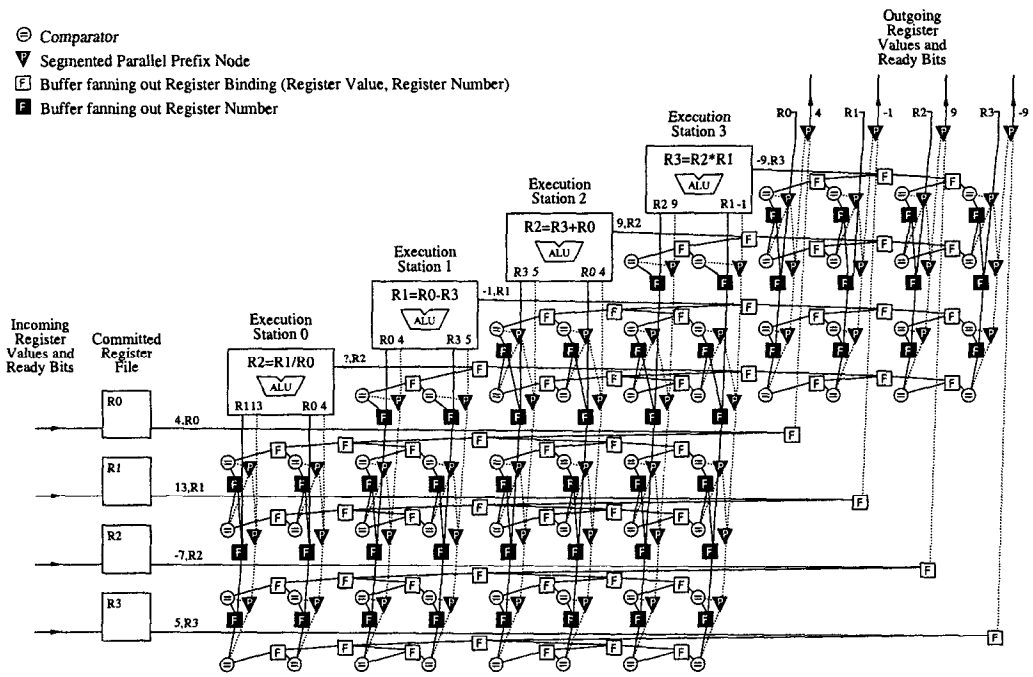

Figure 8: An Ultrascalar II datapath with four instructions per cluster and four logical registers. The datapath has been reimplemented with fan-out trees and parallel prefix circuits in order to reduce the gate delay to $O(\log(n + L))$.

into the register file (lower left corner). The stations refill with new instructions, and computation resumes. The memory switch will be discussed in Section 5.

Each execution station sends the numbers of the two registers it needs downward. Each execution station sends rightward the number of the register to which it writes. At each cross-point, the register numbers are compared to determine if a particular row provides the value being requested by the station in a particular column. The value from the highest such matching row is propagated upward to the requesting station.

Thus, the Ultrascalar II network shown in Figure 7 consists of columns of comparators and multiplexers. Each column searches through rows of register bindings (number, value, and ready bit) for the nearest matching register number. The value and the ready bit of the nearest match propagate back up the column. For example, the left column of Station 3, searches for the argument $R2$. The path followed by the Register 2 data is shown in bold. The very first comparison in the column succeeds because the instruction in Station 2 writes $R2$. Since Station 2 has finished computing, the column returns the value 9 and a high ready bit. Note that the column ignores the earlier, unfinished write to $R2$ by Station 0; allowing Station 3 to issue out of order. Also shown in bold is the path that the Register 1 data follows to get from Station 1 to Station 3. Note that the result from Station 1 also propagates to the R1 column of the outgoing register values (but the figure does not show the path in bold.) Similarly, the data coming out of the right of Station 3 moves to the rightmost column, where it is muxed and produced as an outgoing register value.

The datapath in Figure 7 has linear asymptotic gate delay. During each clock cycle, the column of the last station serially searches for the station's argument among all the preceding instructions' results and all the initial register values. As a result, the clock period grows as $O(n + L)$ (where, again, n is the number of outstanding instructions and L is the number of logical registers.) The gate delay can be reduced to logarithmic from linear by converting the Ultrascalar II network into a mesh-of-trees network. (For a discussion of mesh-of-trees layouts, see [9].) Instead of broadcasting register numbers, values, and ready bits horizontally to as many as $2n + L - 2$ columns, we fan them out through a tree of buffers (i.e., one-input gates that compute the identity.) Similarly, instead of broadcasting each argument's register number downward to as many as $n + L - 1$ comparators in its column, we fan out the request through a tree of buffers. Once all values have fanned out in $O(\log(L + n))$ gate delay, each comparator carries out its comparison in additional $O(\log \log L)$ gate delay. Finally, we replace each column of multiplexers by a (noncyclic) segmented parallel prefix circuit with the associative operator $a \otimes b = a$, with the result of each comparison serving as the segment bit, and with the fanned out register value and ready bit serving as the input. Each parallel prefix tree returns the most recent value and ready bit of its column's register at the root in an additional $O(\log(n+L))$ gate delay. (The tree circuits used here are more properly referred to as reduction circuits rather than parallel-prefix circuits.) Figure 8 shows our original linear-gate-delay datapath converted to a logarithmic-gate-delay datapath. The nodes of the prefix trees have been marked with a P, the nodes of the fan-out trees with an F.

While the register datapath of the Ultrascalar II is very different from the Ultrascalar I's, the two processors have some common ground. Just like the Ultrascalar I, the Ultrascalar II connects stations to memory by fat trees and enforces instruction dependencies using several instances of the CSPP circuit from Figure 5. Both processors rename registers, forward new results in one clock cycle, issue instructions out of order, and revert from branch misprediction in one clock cycle. The Ultrascalar II as described is less efficient than the Ultrascalar I because its datapath does not wrap around. As a result, stations idle waiting for everyone to finish before refilling. The Ultrascalar II can easily be modified to handle wrap-around, but it makes the explanation needlessly complex without improving the asymptotics. Furthermore, it appears to cost nearly a factor of two in area to implement the wrap-around mechanism. In addition, as we shall see, the Ultrascalar II without wraparound is a useful component in the hybrid processor.

5 Ultrascalar II Floorplan and Analysis

The previous section has described the Ultrascalar II datapath and analyzed its asymptotic gate delay. This section provides a layout and analyzes the wire delay and area.

The side length of the Ultrascalar II is straightforward to compute. Figure 7 shows our floorplan of the linear-gate-delay Ultrascalar II. The execution stations and the register datapath are layed out just as was shown in Figure 7. That is, the execution stations are layed out along a diagonal, with the register datapath layed out in the triangle below the diagonal. The memory switches are placed in the space above the diagonal. Since $M(n) = O(n)$ in all cases, there is always space in the upper-triangle for all the memory switches with at worst a constant blowup in area. Thus, the entire Ultrascalar II can be layed out in a box with side-length $\Theta(n + L)$.

Note, however, that if the tree-of-meshes implementation is used to reduce the number of gate delays, then the side length increases to $\Theta((n+L)\log(n+L))$. One can use a mixed strategy in which one replaces the part of each tree near the root with a linear-time prefix circuit. This works well in practice because at some point the wire-lengths near the root of the tree become so long that the wire-delay is comparable to a gate delay. At that point, there is no asymptotic penalty for increasing the number of gate delays. This scheme's asymptotic results are exactly the same as for the linear-time circuit (the wire delays, gate delays, and side length are all n) with greatly improved constant factors. (In our VLSI implementation, we found that there was enough space in our Ultrascalar II datapath to implement about three levels of the tree without impacting the total layout area, since the gates were dominating the area.)

6 Ultrascalar Hybrid

Our third processor is a hybrid of the Ultrascalar I and Ultrascalar II. The hybrid obtains the advantages of the Ultrascalar II for small processors and the advantages of the Ultrascalar I for large processors. The processor is divided into clusters, each containing C stations. The cluster is implemented using the Ultrascalar II datapath, and then the clusters are connected together using the Ultrascalar I datapath. Throughout this section, we will use the linear-gate-delay grid-like datapath of Figure 7. The results easily extend to the log-depth tree-of-meshes datapath of Figure 8.

Figure 10 shows the floorplan of a 32-instruction hybrid Ultrascalar processor ($n = 32$) with full memory bandwidth ($M(n) = \Theta(n)$.) The processor implements an instruction set with eight logical registers ($L = 8$.) The floorplan contains four clusters, each containing 8 stations ($C = 8$). Thus, each cluster is an 8-instruction Ultrascalar II. As we shall see, it is not a coincidence that $C = L$. The four clusters are connected together via an Ultrascalar I datapath. In order to present the correct interface to the Ultrascalar I datapath, each Ultrascalar II cluster has been slightly expanded to generate modified bits, as shown in Figure 9. Each cluster now generates a modified bit for each logical register using either a series of OR gates or a tree of OR gates.

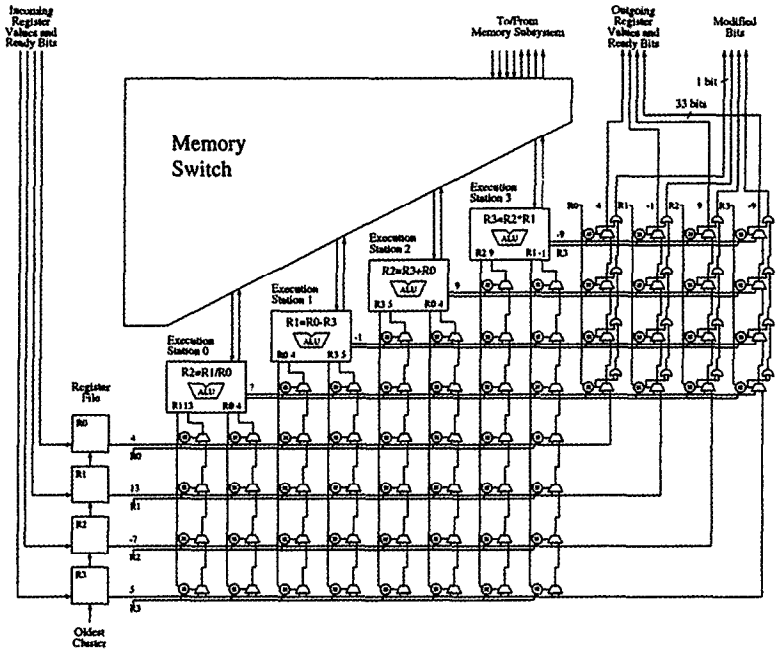

Figure 9: The cluster for the Hybrid Ultrascalar. The Hybrid Ultrascalar cluster includes OR gates to generate modified bits, whereas Figure 7 shows an Ultrascalar II cluster which has no modified bits. This figure shows a cluster of size 4 instead of size 8 so that the circuitry can be more easily seen.

Two instructions communicate their results, one to the other, as follows. If the two instructions happen to be placed in the same cluster, then they use the grid-like Ultrascalar II datapath. The two instructions may not always end up in the same cluster. If there are more than 6 instructions in the dynamic sequence between the two communicating instructions, then it must be the case that the two instructions are placed in different clusters. Even if one of the instructions immediately follows the other, they can be placed in different clusters. For example, the first instruction may be placed in Station 7 of Cluster 0, and the next instruction may be placed in Station 0 of Cluster 1. In this case, the data from the first instruction travels to the outgoing register values via the grid-like datapath, and then it travels through the tree-like Ultrascalar I datapath to get from Cluster 0 to Cluster 1, and then it travels through Cluster 1's grid-like datapath to get to its destination.

From the viewpoint of the Ultrascalar I part of the datapath, a single cluster behaves just like a subtree of 8 stations in the ordinary Ultrascalar I circuit. The cluster executes up to 8 contiguous instructions in parallel, and produces up to 8 new values with the modified bits set. Alternatively, we can view a cluster as taking on the role of a single "super" execution station that is responsible for running 8 instructions instead of only one. In this view, each cluster behaves just like an execution station in the Ultrascalar I. Just like in the Ultrascalar I, exactly one cluster is the oldest on any clock cycle, and the commited register file is kept in the oldest cluster. This works by having all clusters other than the oldest latch incoming registers into their register files, while the oldest cluster inserts the initial register file into the datapath. As before, newly written results propagate to all readers in one clock cycle.

The hybrid can be analyzed as follows. For the hybrid processor, containing C execution stations and implemented with linear gate delay, it will turn out that the optimum value of C is about equal to L. The side-length of a n-wide hybrid processor, $U(n)$, is expressed by a recurrence that is similar to the Ultrascalar I recur-

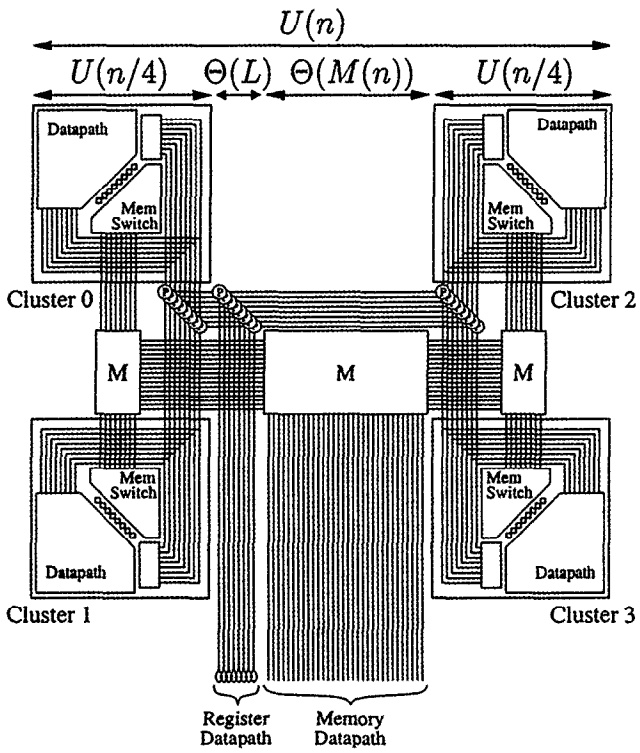

Figure 10: The floorplan of our Hybrid Ultrascalar with four clusters of eight instructions each, and with full memory bandwidth.

rence:

$$U(n) = \begin{cases} \Theta(n+L) & \text{if } n \leq C, \\ \Theta(L+M(n)) + 2U(n/4) & \text{if } n \geq C. \end{cases}$$

For $n \geq C$, this recurrence has solution

$$U(n) = \Theta\left(M(n) + L\sqrt{\frac{n}{C}} + \sqrt{nC}\right).$$

To find the value of C that minimizes $U(n)$, one can differentiate and solve for $dU/dC(n) = 0$, to conclude that the side-length is minimized when $C = \Theta(L)$. (Thus, our decision to set the cluster side-length to L in our implementation is close to the optimal cluster size.) When $C = \Theta(L)$ we have

$$U(n) = \Theta(M(n) + \sqrt{nL}).$$

This is optimal as a function of M and existentially tight as a function of n and L. (There are cases of n and L for which it can be proved that the side length must be $\Omega(\sqrt{nL})$. It seems likely that this bound is universally tight, but we have not proved it.) It is straightforward to implement and analyze a hybrid using clusters that have only logarithmic delays.

7 Analytical and Empirical Comparison

Here we present both an analysis and an empircial study comparing the complexity of the three processors.

Figure 11 shows a summary of the results for the various processors. The analysis shows that the hybrid dominates the other processors. The Ultrascalar I and Ultrascalar II are incomparable, each beating the other in certain cases. For example, for smaller processors ($n \leq O(L^2)$) the Ultrascalar II dominates the Ultrascalar I by a factor of $\Theta(L/\sqrt{n})$, but for larger processors the Ultrascalar I dominates the Ultrascalar II. In fact, for large processors ($n = \Omega(L)$) with low memory bandwidths ($M(n) = O(n^{1/2-\epsilon})$) the Ultrascalar I wire delays beat the Ultrascalar II by a factor of $\sqrt{n}/L$, and the hybrid beats the Ultrascalar I by an additional factor of $\sqrt{L}$.

Our analytical results show that memory bandwidth is the dominating factor in the design of large-scale processors. If processors require memory bandwidth linear in the number of outstanding instructions ($M(n) = O(n)$), the wire delays must also grow linearly. In this case, all three processors are asymptotically the same. Memory bandwidth does not even have to grow linearly to cause problems, however. If it grows as slowly as $\Omega(n^{1/2} + \epsilon)$, it can dominate the design. One way to reduce the bandwidth requirements may be to use a cache distributed among the clusters. The memory bandwidth pressure can also be reduced by using memory-renaming hardware, which can be implemented by CSPP circuits. With the right caching and renaming protocols, it is conceivable that a processor could require substantially reduced memory bandwidth, resulting in dramatically reduced chip complexity.

How does the Ultrascalar scale in three dimensions? It depends on what is meant by "three-dimensions". For a two-dimensional chip packaging technology in which there are pads spread throughout the chip, but in which the chip is packaged on a two-dimensional board, the bounds are essentially unchanged. In this situation, the pins are more like inter-layer vias rather than a true three dimensional technology.

In a true three-dimensional packaging technology, the number of wires stacked in the third dimension is comparable to the number of wires in the other dimensions. Three-dimensional technology has been used at the board level (daughter boards are placed on the motherboards of many computer systems).

In a true three-dimensional packaging technology the Ultrascalar bounds do improve because, intuitively, there is more space in three dimensions than in two. The recurrence relations are similar to the ones above, and they can be similarly solved. An Ultrascalar I with small memory bandwidth can be layed out in volume $nL^{3/2}$ with wire lengths $n^{1/3}L^{1/2}$, whereas large memory bandwidth ($M(n) = \Omega(n^{2/3+\epsilon})$) requires requires an additional volume of $\Theta((M(n))^{3/2})$. This can be understood at an intuitive level by noting that for large memory bandwidth, the surface area of the bounding box for n stations must have area at least $\Omega(M(n))$, and hence the side length must be $\Omega((M(n))^{1/2})$.

The Ultrascalar II requires volume only $O(n^2+L^2)$ whether the linear-depth or log-depth circuits are used, whereas in two dimensions an extra $\log^2 n$ area is required to achieve log-depth circuits.

For the hybrid, in systems with small memory bandwidth, the optimal cluster size is $\Theta(L^{3/4})$, as compared to $\Theta(L)$ in two dimensions. The total volume of the hybrid is $\Theta(nL^{3/4})$, as compared to an area of $\Theta(nL)$ in two dimensions.

To study the empirical complexity of this processor, our research group implemented VLSI layouts of the Ultrascalar I, the Ultrascalar II, and the hybrid register datapaths using the Magic design tools [11]. Our layouts for varying numbers of outstanding instructions confirm the scaling properties of the various processors. Moreover, the layouts demonstrate the practical advantages of the hybrid for foreseeable values of n and L.

We have chosen to implement a very simple RISC instruction set architecture. Our architecture contains 32 32-bit logical registers. It does not implement floating point operations. Each instruction in the architecture reads at most two registers and writes at most one.

In order to generate processor VLSI layouts in an academic setting, we had to optimize for speed of design. Instead of designing at transistor-level, we based our designs on a library of CMOS standard cells that we wrote. Thus the transistors have not been ideally sized for their respective jobs. For the most part, we did not worry about the optimal size of our gates or the thickness of our wires since these factors are comparable in the two designs. Our academic design tools also limited us to only three metal layers. As a result, we believe that a team of industrial engineers could significantly reduce the size of our layouts. The relative sizes of the layouts should not change much, however.

Figure 12 shows register datapath layouts of the two processors and their relative size. The layouts implement communication among instructions; they do not implement communication to memory. Figure 12(a) shows a 64-instruction-wide Ultrascalar I register datapath corresponding in functionality to Figure 4. Figure 12(b) shows a 128-instruction-wide 4-cluster hybrid Ultrascalar register datapath. Both processors use the same ALUs. The same H-tree interconnect that connects execution stations in the Ultrascalar I also connects clusters in the hybrid.

Both layouts are implemented in a 0.35 micrometer CMOS technology with three layers of metal. (Today's real 0.35 micrometer technologies typically have more than three layers of metal.) The Ultrascalar I datapath includes 64 processors in an area of 7 cm × 7 cm, which is 13,000 processors per square meter. The hybrid datapath includes 128 processors in an area of 3.2 cm × 2.7 cm, which is 150,000 processors per square meter (about 11.5 times denser.)

The details of the hybrid datapath as implemented with Magic differ somewhat from the floorplan shown in Figure 10. A large fraction of the area of a cluster shown in Figure 10 is used to move the incoming registers to the same edge of the cluster as the outgo-

	Ultrascalar I	Ultrascalar II (linear gates)	Ultrascalar II (log gates)	Hybrid ($n = \Omega(L)$) (linear-gate clusters)
		$M(n) = O(n^{1/2-\epsilon})$		
Gate Delay	$\Theta(\log n)$	$\Theta(n+L)$	$\Theta(\log(n+L))$	$\Theta(L+\log n)$
Wire Delay	$\Theta(\sqrt{nL})$	$\Theta(n+L)$	$\Theta((n+L)\log(n+L))$	$\Theta(\sqrt{nL})$
Total Delay	$\Theta(\sqrt{nL})$	$\Theta(n+L)$	$\Theta((n+L)\log(n+L))$	$\Theta(\sqrt{nL})$
Area	$\Theta(nL^2)$	$\Theta(n^2+L^2)$	$\Theta((n+L)^2\log^2(n+L))$	$\Theta(nL)$
		$M(n) = \Theta(n^{1/2})$		
Gate Delay	$\Theta(\log n)$	$\Theta(n+L)$	$\Theta(\log(n+L))$	$\Theta(L+\log n)$
Wire Delay	$\Theta(\sqrt{n}(L+\log n))$	$\Theta(n+L)$	$\Theta((n+L)\log(n+L))$	$\Theta(\sqrt{nL})$
Total Delay	$\Theta(\sqrt{n}(L+\log n))$	$\Theta(n+L)$	$\Theta((n+L)\log(n+L))$	$\Theta(\sqrt{nL})$
Area	$\Theta(n(L^2+\log^2 n))$	$\Theta(n^2+L^2)$	$\Theta((n+L)^2\log^2(n+L))$	$\Theta(nL)$
		$M(n) = \Omega(n^{1/2+\epsilon})$		
Gate Delay	$\Theta(\log n)$	$\Theta(n+L)$	$\Theta(\log(n+L))$	$\Theta(L+\log n)$
Wire Delay	$\Theta(\sqrt{nL}+M(n))$	$\Theta(n+L)$	$\Theta((n+L)\log(n+L))$	$\Theta(\sqrt{nL}+M(n))$
Total Delay	$\Theta(\sqrt{nL}+M(n))$	$\Theta(n+L)$	$\Theta((n+L)\log(n+L))$	$\Theta(\sqrt{nL}+M(n))$
Area	$\Theta(nL^2+(M(n))^2)$	$\Theta(n^2+L^2)$	$\Theta((n+L)^2\log^2(n+L))$	$\Theta(nL+(M(n))^2)$

Figure 11: A comparison of the results for the various processors.

ing registers. In our Magic layout we used additional metal layers to route the wires for the incoming registers over the datapath instead, saving that area. We also did not lay out the memory datapath, however we left space in the design for a small datapath of size $M(n) = O(1)$. In addition, the ALUs do not fit along the diagonal as tightly as shown in the floorplan. We placed the 32 ALUs of each cluster in 4 columns of 8 ALUs each, arrayed off the diagonal.

One way to improve the overall performance of all the processors is to change the timing methodology. We described, in this paper, processors that use a global single-phase clock with all communications between components being completed in one clock cycle. For each of the three processors, it is possible to pipeline the system, however, so that the long communications paths would include latches. Taking this idea to an extreme, all three processors could be operated in a self-timed fashion. Understanding the overall performance improvement of such schemes will require detailed performance simulations, since some operations, but not all, would then run much faster. A back-of-the envelope calculation is promising however: Half of the communications paths from one station to its successor are completely local. In such a processor, a program could run faster if most of its instructions depend on their immediate predecessors rather than on far-previous instructions.

The Ultrascalar ideas could be realizable in a few years. To make this happen, more effort needs to be spent to reduce the various constant factors in the design. For example, in the designs presented here, the ALU is replicated n times for an n-issue processor. In practice, ALUs can be effectively shared (especially floating-point ALUs), reducing the chip area further. We have shown how to implement efficient scheduling logic for a superscalar processor that shares ALUs [6]. We believe that in a 0.1 micrometer CMOS technology, a hybrid Ultrascalar with a window-size of 128 and 16 shared ALUs (with floating-point) should fit easily within a chip 1 cm on a side.

Acknowledgments

Dr. Christopher Joerg of Compaq's Cambridge Research Laboratory pointed out the trend of increasing numbers of logical registers and argued that we should treat the number of logical registers as a scaling parameter. Yale graduate student Vinod Viswanath layed out the Ultrascalar I datapath in VLSI.

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] William J. Dally and John W. Pulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[3] 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.

[4] Bruce A. Gieseke et al. A 600MHz superscalar RISC microprocessor with out-of-order execution. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'97)*, pages 176–177, February 1997.

[5] Dana S. Henry and Bradley C. Kuszmaul. Cyclic segmented parallel prefix. Ultrascalar Memo 1, Yale University, 51 Prospect Street, New Haven, CT 06525, November 1998. http://ee.yale.edu/papers/usmemo1.ps.gz.

[6] Dana S. Henry and Bradley C. Kuszmaul. An efficient, prioritized scheduler using cyclic prefix. Ultrascalar Memo 2, Yale University, 51 Prospect Street, New Haven, CT 06525, 23 November 1998. http://ee.yale.edu/papers/usmemo2.ps.gz.

[7] 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.

[8] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *The 19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 46–57, Gold Coast, Australia, May 1992. ACM SIGARCH Computer Architecture News, Volume 20, Number 2.

[9] F. Thompson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.

[10] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[11] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. Magic: A VLSI layout system. In *ACM IEEE 21st Design Automation Conference*, pages 152–159, Los Angeles, CA, USA, June 1984. IEEE Computer Society Press.

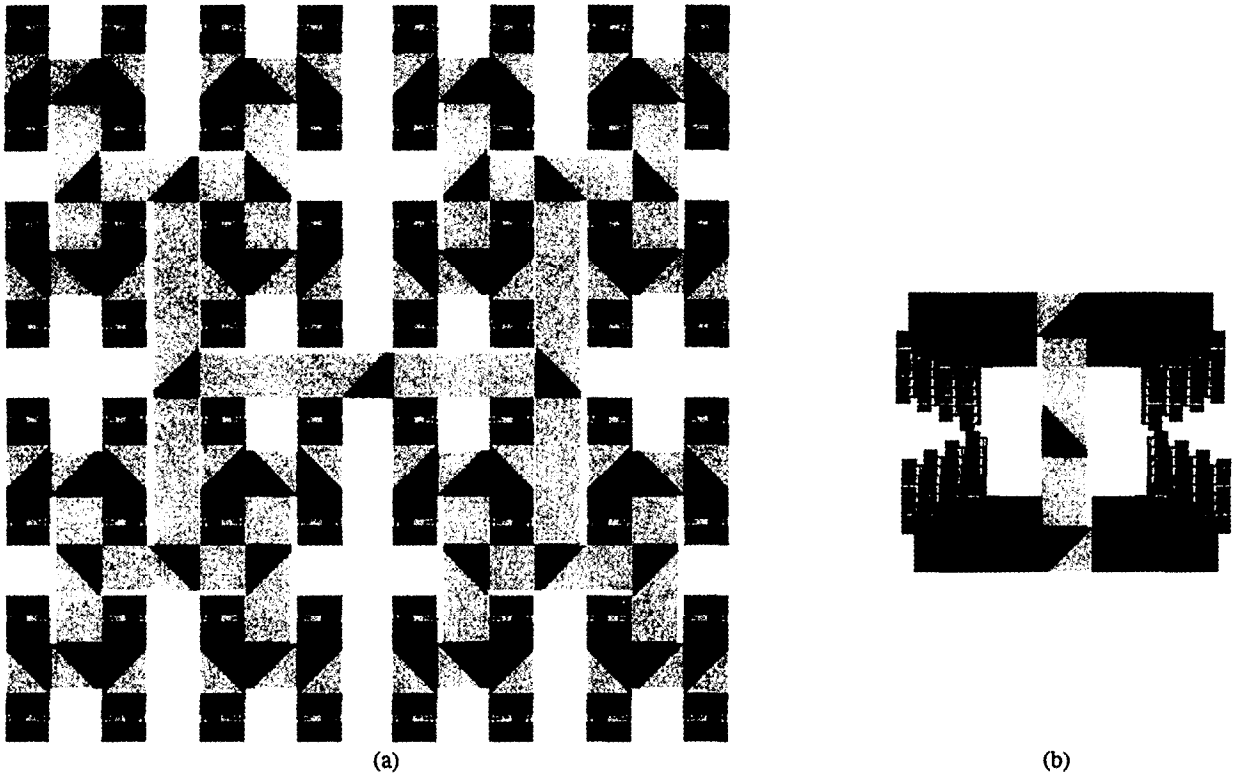

Figure 12: (a) A 64-instruction-wide Ultrascalar I register datapath. (b) A 128-instruction-wide 4-cluster hybrid Ultrascalar register datapath.

[12] 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.

[13] Sanjay Jeram Patel, Marius Evers, and Yale N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 262–271, Barcelona, Spain, 27 June–1 July 1998. IEEE Computer Society TCCA and ACM SIGARCH, IEEE Computer Society, Los Alamitos, CA, published as *Computer Architecture News*, 26(3), June 1998. http://www.eecs.umich.edu/HPS/pub/promotion_isca25.ps.

[14] 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.

[15] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO 29)*, pages 24–34, Paris, France, 2–4 December 1996. http://www.cs.wisc.edu/~ericro/TC_micro29.ps.

[16] Richard L. Sites, Richard T. Witek, et al. *Alpha Architecture Reference Manual.* Digital Press, Boston, MA, third edition, 1998.

[17] Peter A. Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for lisp. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[18] TI's 0.07-micron CMOS technology ushers in era of gigahertz DSP and analog performance. http://www.ti.com/sc/docs/news/1998/98079.htm, 26 August 1998. Accessed April 4, 1999.

[19] D. J. Wheeler. Programme organization and initial orders for EDSAC. *Proc. of the Royal Soc.*, 202:573–589, 1950.

[20] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 67–76, Tokyo, Japan, 20–22 July 1993. ACM SIGARCH.