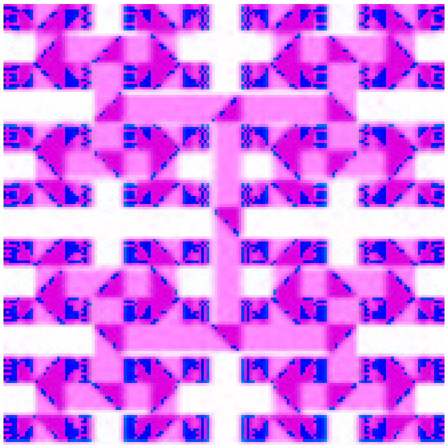
Ultrascalar Memo 3



The Ultrascalar Processor

An Asymptotically Scalable Superscalar Microarchitecture

or How I Learned to Stop Worrying and Love Out-of-Order Execution[a]

Dana S. Henry, Bradley C. Kuszmaul, and Vinod Viswanath
Yale University
{dana,bradley,vinod}@ee.yale.edu

December 30, 1998

[a] To appear in The Twentieth Anniversary Conference on Advanced Research in VLSI (ARVLSI'99), Atlanta, GA, March 21–24, 1999.

1 Introduction

Today's superscalar processors 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 and irregular, requiring much effort and ingenuity to implement well. Furthermore, the delays through many of the circuits grow quadratically with issue width (the maximum number of simultaneously fetched or issued instructions) and window size (the maximum number of instructions within the processor core), making future scaling of today's designs problematic [11, 4, 5]. With billion transistor chips on the horizon,[1] this scalability barrier appears to be one of the most serious obstacles for high-performance uniprocessors in the next decade. Surprisingly, it is possible to extract the same instruction-level parallelism (ILP) with a regular circuit structure that has only logarithmic gate delay and linear wire delay (speed-of-light delay) or even sublinear wire delay, depending on how much memory bandwidth is required for the processor. This paper describes a new processor microarchitecture, called the *Ultrascalar* processor, based on such a circuit structure.

The goal of this paper is to illustrate that processors can scale well with issue width and window size. We have designed a new microarchitecture and layed out its datapath. We have analyzed the asymptotic growth and empirically computed its area and critical-path delays for different window sizes. We have not optimized the Ultrascalar architecture to be competitive with today's designs. Although we outline design choices that could make the Ultrascalar competitive, an optimized processor design is outside the scope of this paper. This paper also does not evaluate the benefits of larger issue widths and window sizes. Some work has been done showing the advantages of high-issue-width and high-window-size processors. Lam and Wilson suggest that ILP of ten to twenty is available with an infinite instruction window and good branch prediction [7] . 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]. The amount of parallelism available in a thousand-wide instruction window with realistic branch prediction, for example, is not well understood however. The ultimate value of the Ultrascalar microarchitecture will depend on careful engineering for specific window size and on the available parallelism in programs.

Microprocessor Performance

The standard model for modeling the performance of a microprocessor [6] says that the time to run a program is $T = N \cdot \text{CPI} \cdot \tau$ where

- N is the number of instructions needed to run the program,
- CPI is the number of clock periods per instruction, and
- τ is the length of a clock period in seconds, i.e. the cycle time.

[1] Texas Instruments announced recently a 0.07 micron process with plans to produce processor chips in volume production in 2001 [19].

The value of τ is determined by the critical-path length through any pipeline stage, that is the longest propagation delay through any circuit measured in seconds. Propagation delay consists of delays through both gates and wires, or alternately of delays through transistors driving RC networks. We are not changing N or directly changing CPI, but rather we aim to reduce the clock cycle by redesigning the processor to use circuits with reduced critical-path length.

An alternate way to avoid slowing down the clock is by breaking down the processor into more pipeline stages. Increasing the number of pipeline stages offers diminishing returns, however, as pipeline registers begin to take up a greater fraction of every clock cycle and as more clock cyles are needed to resolve data and control hazards. In contrast, shortening the critical path delay of the slowest pipeline stage translates directly into improved program speed as the clock period decreases and the other two parameters remain unchanged.

The critical-path delays of many of today's processor circuits do not scale well. For example, Palacharla, Jouppi, and Smith [11] find that many of the circuits in today's superscalars have asymptotic complexity $\Omega(I^2 + W^2)$, where I is the issue width and W is the window size of the processor. For today's processors, optimized for I equal to four, and W in the range of 40 to 56, the delays appear to be practically linear, however, although the quadratic terms appear to become important for slightly larger values of I and W.[2] Increasing issue widths and increasing window sizes are threatening to explode the cycle time of the processor. In contrast, all of the Ultrascalar processor circuits grow much more slowly with gate delays of $O(\log I + \log W)$ and wire delays of $O(\sqrt{I} + \sqrt{W})$ for memory bandwidth comparable to today's processors. Due to the constants involved, the Ultrascalar may not be as fast as today's quadratic-time superscalars for today's values of W and I. We believe that with some engineering effort the crossover point where Ultrascalar becomes faster may be as low as $I = 8$ and $W = 64$, however. The asymptotic advantage of the Ultrascalar over today's circuits translates to perhaps an order-of-magnitude or more advantage when W is on the order of several hundreds or thousands, a design point advocated by [14].

The Ultrascalar processor breaks the scalability barrier by completely restructuring the microarchitecture of the processor. The Ultascalar turns the processor's datapath into a logarithmic depth network that efficiently passes data from producer instructions to consumer instructions within the reordering window. The network eliminates the need for separate renaming logic, wake-up logic, bypass logic, and multi-ported register files.

The rest of this paper is organized as follows. Section 2 explains the mechanisms within the Ultrascalar processor core and analyzes the circuits' performance in terms of gate delays. Section 3 briefly explores the space of memory subsystems that can be attached to the Ultrascalar processor core. Section 4 analyzes the performance of the Ultrascalar circuitry in more detail, taking into account the layout to produce wire delay and area bounds. Section 5 presents empirical delay and area data derived from our layouts and suggests ways to improve the constants. Section 6 concludes by comparing the Ultrascalar to other work exploiting large ILP and discussing other applications of the Ultrascalar circuits.

2 The Ultrascalar Core

This section describes the core of the Ultrascalar processor. The Ultrascalar processor core performs the same functions as a typical superscalar processor core. It renames registers, analyses register and memory data dependencies, executes instructions out of order, forwards results, efficiently reverts from mispredictions, and commits and retires instructions. The Ultrascalar processor core is much more regular and has lower asymptotic critical-path length than todays superscalars, however. In fact, all the scaling circuits within the processor core are instances of a single algorithm, parallel prefix, implemented in VLSI. Because of the core's simplicity, it is easily apparent how the number of gates within a critical path grows with the issue width and window size.

The core does not include the memory and branch prediction subsystems. Instead the core presents the same interface to the instruction fetch unit and the data cache as today's superscalar processor cores. Predicted instruction sequences enter the core, and data load and store requests are initiated by the core. We briefly discuss possible memory subsystems in Section 3, but the Ultrascalar core will benefit from the any advances in effective instruction fetch rate and in data memory bandwidth that can be applied to traditional superscalar processors. In particular, since the Ultrascalar processor core performs the same functions as the core of today's superscalars, it achieves the same CPI performance as existing superscalars when attached to a traditional 4-instruction-wide fetch unit using traditional branch prediction techniques and a traditional cache organization. As effective fetch rates and data bandwidths

[2] Note that for today's processors with large W the window is typically been broken in half with a pipeline delay being paid elsewhere. An HP processor sets $W = 56$ [5]. The DEC 21264 sets $W = 40$ [4]. Those systems employ two windows, each half size, to reduce the critical-path length of the circuits. Communicating between the two halves typically requires an extra clock cycle.

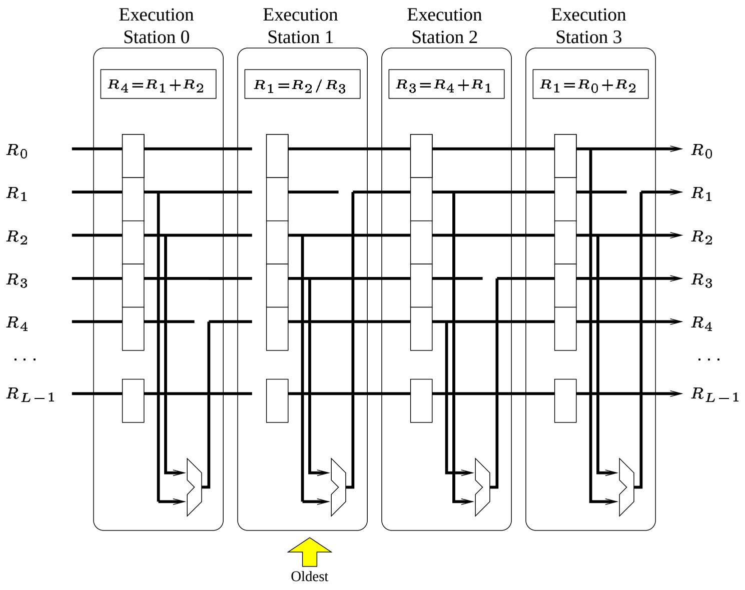


Figure 1: A linear-time datapath for the Ultrascalar processor. This simplified network shows execution stations connected by a pipelined ring network: the signals labeled R_i at the right are connected to the corresponding signals on the left.

increase, the Ultrascalar core can scale gracefully, raising the CPI without exploding the cycle time.

A Datapath with Linear Gate Delays

One way to think of the Ultrascalar processor is that it uses a circuit-switched network to compile at runtime the dataflow graph for a dynamic sequence of unrenamed instructions. The circuit switched network connects producer instructions with consumer instructions. The structure and the layout of this network are the key to the Ultrascalar's scalable performance.

Before we introduce the Ultrascalar, let us briefly consider a simpler datapath in which the network connecting producer instructions to consumer instructions consists of a simple ring. We will refer to this datapath as the *linear-time* datapath. Figure 1 illustrates. The ring network routes the values of L logical registers[3] through a pipelined series of execution stations. Each execution station holds and eventually executes one instruction from a dynamic sequence of instructions. Executing an instruction may take only one clock cycle (e.g., for integer addition) or many clock cycles (e.g., for division.) The number of execution stations corresponds to the number of outstanding instructions within the processor much like the instruction window in today's superscalars. As in today's superscalars, the fetch width is independent of the number of outstanding instructions. Newly fetched instructions simply refill execution stations in a wrap-around fashion, starting with the oldest instruction. In the figure, the oldest instruction in the current sequence resides in Execution Station 1, the youngest in Execution Station 0. Note that the pipeline registers of Execution Station 1 hold the committed state of the register file. Throughout the datapath, each register value has a ready bit. The ready bit is associated with the wires carrying the register value and indicates whether the value has already been computed. As instructions complete, they retire from the datapath and new instructions enter the datapath. Eventually, the output wires of Execution Station 0 hold the new state of the register file, with all of their ready bits set to high.

Consider the performance of the linear-time datapath example in Figure 1. The complete sequence of instructions currently in the datapath (with the corresponding execution station shown to the right) is:

Instruction	Execution Station
$R_1 = R_2/R_3$	(1)
$R_3 = R_4 + R_1$	(2)
$R_1 = R_0 + R_2$	(3)

[3] The value of L is determined by the instruction set architecture. For example, for many RISC architectures $L = 32$.

$R_4 = R_1 + R_2$ (0)

Suppose that division takes 24 clocks and addition takes one.

- On Clock 0, we assume that all the registers become valid at Station 1. Station 1 begins executing.
- On Clock 1, all the registers except for R_1 become valid at Station 2. Station 2 waits for R_1.
- On Clock 2, all the registers except for R_1 and R_3 become valid at Station 3. Station 3 executes and produces a new value of R_1.
- On Clock 3, all the registers except for R_3 become valid at Station 0. Station 0 executes and produces a new value of R_4.
- On Clock 23, Station 1 finishes executing.
- On Clock 24, Station 2 executes.
- On Clock 26, all the registers become valid at Station 0.

Note that the instructions executed out of order. The last two instructions completed long before the first two. Moreover, the datapath automatically renamed registers. The last two instructions did not have to wait for the divide to complete and write R_1.

Our linear-time datapath bears similarities to the counterflow pipeline [18]. Like counterflow, the linear-time datapath automatically renames registers and forwards results. Counterflow provides a mechanism for deep pipelining, rather than large issue width, however. The counterflow pipeline is systolic, with instructions flowing though successive stages of the datapath. Since instructions serially enter the pipeline in the first pipeline stage, the CPI is limited by the rate at which instructions can enter the pipeline. It is not clear how counterflow could be modified to increase its issue width. In contrast, in our linear-time datapath, all execution stations can refill with new instructions simultaneously. (We will discuss how to implement simultaneous refill in Section 3.) Thus, whereas our linear datapath has no corresponding limit on CPI (and is limited by the clock period), the counterflow can push the clock period down but is limited to one CPI.

Another difference between our linear datapath and counterflow is that counterflow uses less area to route data. It only passes results and arguments down its pipeline, not the entire register file. In Section 5 we will discuss a similar modification to reduce the number of wires used in the Ultrascalar.

One weakness shared by counterflow as well as our linear-time datapath is the speed of routing. In a synchronous implementation our linear-time datapath, if a producer and a consumer of a register value are separated by n instructions in the dynamic instruction sequence, it takes n clocks to serially route the value through all n intermediate execution stations. For example, it took 3 clocks to route R_2 from Station 1 to Station 0. (In a counterflow pipeline, it typically would take $n/2$ clocks to perform this operation.) This linear delay can be intolerably high compared to today's superscalars that forward values within one clock cycle.

The Ultrascalar Datapath with Logarithmic Gate Delays

The Ultrascalar datapath replaces the linear-time ring network of Figure 1 with a faster logarithmic-depth network performing the same function. Figure 2 illustrates the Ultrascalar datapath. The logarithmic-depth network forms a bidirectional tree that routes the values of all L logical registers among the execution stations. The execution stations are the same as for the linear-time datapath, except that they produce an additional one-bit "modified" output for each logical register. This "modified" bit will be explained shortly. As before, instructions are assigned to execution stations in a wrap-around sequence.

Since the routing of each of the L registers is independent, it is often easier to consider a single slice, responsible for the routing of one register, of the network. Thus the network in Figure 2 can be broken into L network slices such as the one in Figure 3. To make the example more interesting, we have increased the number of execution stations in Figure 3 to eight. Each slice routes the values of one register, say R_5, among the eight execution stations. The slice hands each execution station the value of R_5 and its ready bit. The execution station hands back to the slice a potentially new value of R_5, its ready bit, and an additional "modified" bit telling the slice whether the station modifies R_5. The additional modified bit tells the network how to route register values efficiently. For example, if Execution Station 5 holds the oldest instruction and Execution Station 1 modifies R_5 then the values of R_5 will be routed as indicated in Figure 3. The value of R_5 held by Station 5 appears at the inputs of Stations 6, 7, 0, and 1. The value of R_5 produced by Station 1 appears at the inputs of Stations 2–5. Notice that Station 5 which is holding the oldest instruction also set its modified bit to 1, telling the network that it has in effect modified R_1.

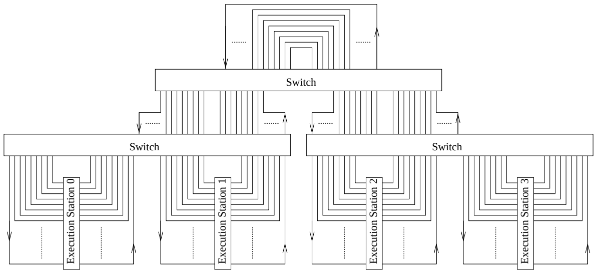


Figure 2: A high-level view of the logarithmic-depth network connecting all the execution stations of the Ultrascalar.

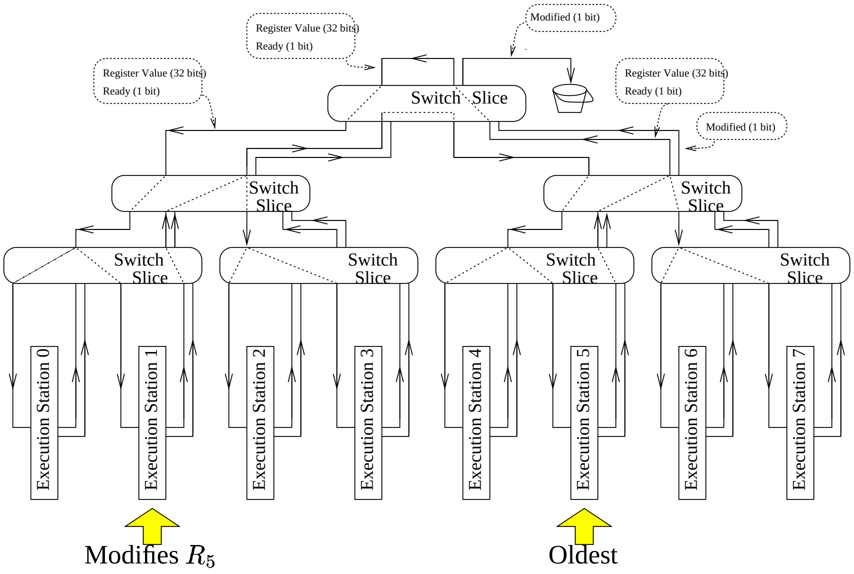


Figure 3: One slice of the logarithmic-depth network. One slice propagates the value of one register, say R_5, from producer instructions to consumer instructions.

Figure 4a shows the circuit within each switch slice of Figure 3. The thick lines carry 33 bits (32 for register value and 1 to indicate that the register is ready.) The thin lines carry the “modified” bit up the tree. Note that the critical path through through each slice of the network consists of $(2\lg n - 1)$ multiplexers, where n is the number of execution stations.

The network slice of Figure 3 may appear familiar to some because the slice is an instance of an important parallel algorithm—namely, segmented parallel prefix. In general, a 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. The associative prefix operator[4] in this case is $a \otimes b = a$. Figure 4b shows the generalized circuit within each node of a segmented parallel prefix circuit. The function $\otimes$ can be any associative operator. In addition, we have turned the segmented parallel prefix (spp) 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 a *cyclic*, segmented parallel prefix circuit the preceding nodes wrap around until a node is found whose segment bit is high.

The Ultrascalar datapath shown in Figure 3 routes all available register values to all instructions in $O(\log n)$ gate delays, at the end of each clock cycle. Specifically, it takes $(2\lg n - 1)$ multiplexer delays to route a register value all the way up and down the prefix tree, where n is the number of execution stations. For example, if the number of outstanding instructions is 32, comparable to today’s superscalars, than it takes at most 9 multiplexer delays to route data. Some of this delay can be masked further since the select lines to the multiplexers are available earlier than the

[4] See [2] for a discussion of parallel prefix trees, including the definition of the associative operator.

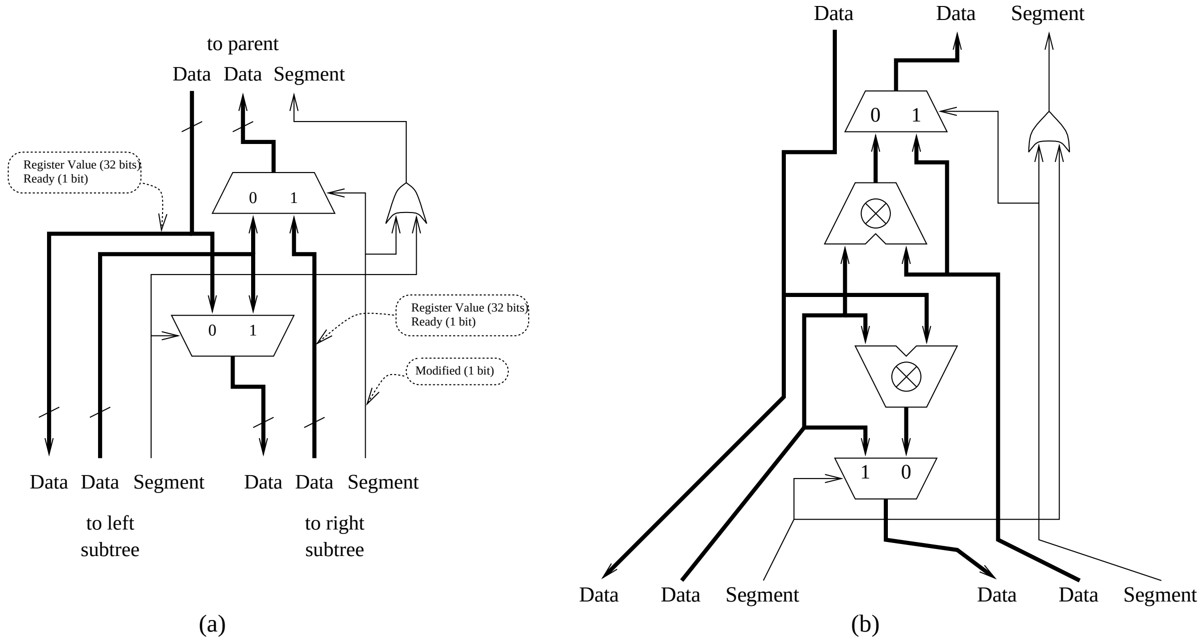


Figure 4: (a) The circuitry inside each switch slice. (b) The cicuitry inside each node of any segmented parallel prefix tree. $\otimes$ can be any associative operation.

values.

Memory Operations

For simplicity, we have avoided showing any memory operations (loads or stores) in the example of Figure 1. Although we will present a more optimized data memory subsystem in Section 3, it is important to point out that the Ultrascalar datapath can use the same memory subsystem as any superscalar processor. From the viewpoint of the memory, the executions stations are indistinguishable from a traditional instruction window.

An execution station cannot read or write the data cache until its memory dependencies have been met. For example, if there are several load instructions between two store instructions, then the loads can run in parallel. But the loads must often wait for at least some previous store operations to complete to get the right data.[5] We first present the design for a conservative strategy, which is to wait for *all* previous store instructions to complete, then we show how to relax this constraint to exploit more parallelism. For the conservative design, we follow the our exposition of the datapath: we first show a linear-delay design for the serializing design, and then convert the design to parallel prefix with log-delay.

Figure 5 shows a linear-time circuit that computes when all previous stores have completed. Each execution station provides a signal O which indicates that the station is the oldest unfinished instruction and a signal T which is true if the station's instruction is not a store, or if it is a completed store (i.e., it is not an uncompleted store). Each station receives a signal P indicating that all previous writes have completed. Note that at all times there must be an execution station providing $\mathsf{O} = 1$, to break the cycle in the priority chain.

Figure 6 shows a log-delay circuit that computes when all previous stores have completed. Again, we are using a cyclic segmented parallel-prefix circuit. The associative prefix operator in this case is $a \otimes b = a \wedge b$.

Now that we have shown how to serialize memory operations without paying very much for computing when the serialization constraint has been satisified. Here we show how to avoid serializing on the completion of memory operations. We observe that in order for a memory operation to execute, it is enough to know that no previous incomplete store is using the same address. Thus, as soon as all previous store memory addresses have resolved, we

[5] Note that if we speculate on the data then the loads need not necessarily wait for the previous stores. Below, we discuss branch speculation, which has similar issues to data speculation.

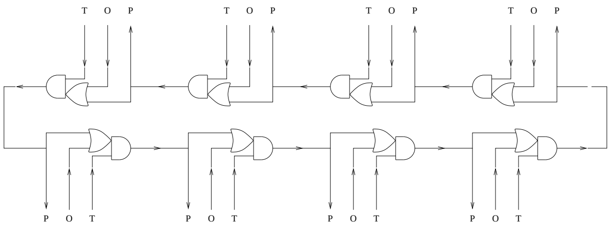


Figure 5: A linear-time circuit that computes when all previous stores have been completed. Each execution station provides a two outputs and one input. Output O ("oldest") indicates that the execution station is the oldest unfinished instruction. Output T ("this write done") indicates that this instruction is not an uncompleted store. Input P ("previous done") indicates that all previous writes have completed.

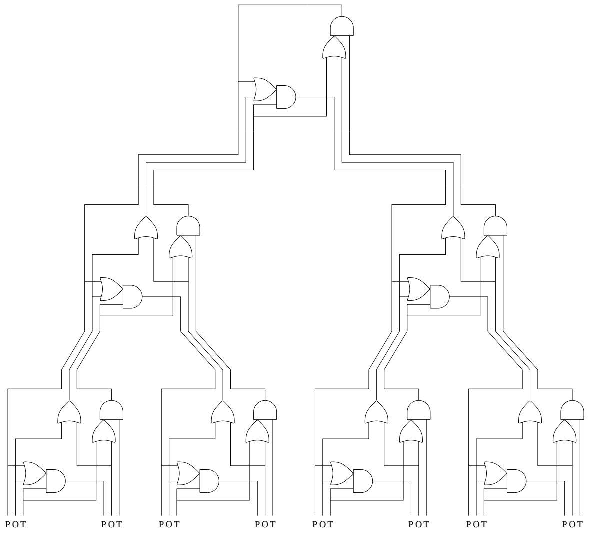


Figure 6: A log-time circuit that computes the same function as the circuit of Figure 5.

can determine whether a particular write may proceed. To do this we let the memory network tell each execution station when its memory operation may proceed.

For the network to compute the memory dependencies, each execution station provides an address and an indication of whether its instruction is a load, a store, or neither. (If the execution station does not yet know, then it should be treated as a store.) The network then informs every execution station when all previous stores that use the same memory address are complete. This works by sending the memory addresses up the network. The memory addresses are kept sorted as they go up the network. At each switch, the two sorted sublists are merged together to a big sorted sublist. If there are any duplicates, then the right child is informed of the duplicate (thus inhibiting the right child from running) using a path that is established during the merging. The duplicates are removed. If a switch higher in the tree notices a conflict, and if there were duplicate users of that address, then both must be notified of the conflict.

Control Flow

A traditional fetch unit can be used to feed instructions to the Ultrascalar datapath. The fetch unit has a write port into every execution station and writes serially predicted instruction blocks into successive execution stations. The fetch unit stops writing when it reaches the oldest execution station.

Which execution station is the oldest can change on every clock cycle, as instructions finish. We can compute which executions station is the oldest, again using the parallel-prefix circuit of Figure 6: In this instance of the circuit, the O bit is the old "oldest" bit (just as for the store completion tree), the T bit indicates that this particular execution station has completed its processing, and the P bit indicates that all previous execution stations have completed. An execution station knows it is the oldest in the next cycle, if it has not finished its own instruction and its incoming P bit is true.

It is very simple to implement speculative execution in the Ultrascalar datapath. When an execution station discovers that its branch instruction has mispredicted, it notifies the fetch unit. The fetch unit starts sending to that unit again, along the right program path. Since each execution station holds the entire register state of the computation, nothing needs to be done to roll back the computation except cause later executions stations execute the correct instructions. Speculation can also be performed on memory operations (speculating that a later load does not depend on an earlier store) or data values using similar mechanisms.

The Ultrascalar datapath described so far exploits exactly the same instruction-level parallel parallelism as one of today's superscalar processors. The Ultrascalar datapath implements renaming, register forwarding, speculation, and dependency checking without requiring multiported register files or other circuits with superlinear critical-path length. Surprising, parallel-prefix trees can perform all the work done by traditional superscalar circuits, with only a logarithmic number of gate delays. Thus, the datapath scales, providing at least as much ILP as today's superscalar processors.

3 Scaling the Memory System

The previous section described a processor core that scales well with increasing numbers of outstanding instructions. In order to exploit ILP, the memory bandwidth too must scale, however. In particular, the processor must be able to issue more loads and stores per clock cycle (i.e. sustain a higher data bandwidth) and the fetch unit must supply more instructions along a correctly predicted program path per clock cycle (i.e. sustain a higher effective instruction bandwidth.) Fortunately, much active research is going on in these areas and the Ultrascalar can benefit from its results. In this section, we review some of the recent work on improving memory bandwidth and suggest additional promising approaches.

Of the two bandwidths, data bandwidth is perhaps less troublesome. To accomodate more loads and stores per clock cycle without scaling up the number of data cache read and write ports, we can resort to the well known mechanism of interleaving. This is the mechanism that we are currently implementing in our layout. The memory subsystem consists of an on-chip level-one cache and an on-chip butterfly network connecting the cache to the execution stations. Much like the main memory in traditional supercomputers [16], the cache is interleaved among a number of banks. In this example, the number of cache banks is the same as the number of execution stations. The nth cache bank holds every nth cache line.[6] The cache banks are connected to the execution stations via a butterfly network and to off-chip memory directly. The butterfly network allows n load and store requests to proceed in parallel if they form a rotation or other conflict-free routing. For example, a vector fetch loop will run without any conflicts if the address increment equals the cache block size. But in general, multiple memory accesses may compete for the same bank, or they may suffer from contention in the butterfly, thus lowering the memory bandwidth, We believe that the same programming, compiler, and hardware techniques used to alleviate bank conflicts in an interleaved main memory will apply to an interleaved data cache.

Another promising approach to increasing data memory bandwidth is to duplicate the data cache. We could allocate a copy of the cache to different subtrees of the Ultrascalar datapath or, in the extreme, to every execution station. Duplicate caches introduce the problem of maintaining coherence between the caches. It is possible that these problems can be resolved in the future using variants of scalable SMP cache-coherence protocols.

Increasing the effective fetch bandwidth poses perhaps a greater problem. To fetch at a rate of much more than one basic block, the fetch unit must correctly predict and supply instructions from several non-consecutive blocks in memory. The mechanisms found in the literature fall into two categories. They either precompute a series of predictions and fetch from multiple blocks or they dynamically precompute instruction traces. In the first category, branch address caches [22] produce several basic block addresses, which are fetched through a highly interleaved cache. In the second category, trace caches [15] allow parallel execution of code across several predicted branches by storing the instructions across several branches in one cache line.

[6] To illustrate, let us assume that the number of banks and execution stations is sixteen, that the cache is direct mapped with block size of one word and total size of 1MB, and that the instruction set architecture uses 32-bit byte addresses. A memory access to address A will then be routed to bank A[5–2]. Bank A[5–2] will lookup entry A[19–6] and compare its tag against address bits A[31-20]. Should the comparison fail, the level-one cache will access off-chip memory.

We propose a parallelized trace cache for the Ultrascalar processor. The on-chip parallelized trace cache is interleaved word by word across cache banks and connected to the execution station by a butterfly network just like the data cache's.[7] The nth cache bank holds the nth word of every trace. An instruction within a trace is accessed by specifying the PC of the first instruction in the trace, the offset within the trace, and some recent branch history information.

The execution stations start fetching from a new trace whenever the old trace ends or a branch within the trace mispredicts. Since each execution station fetches its own instruction from the trace cache, it must know the starting PC of the trace and its instruction's offset within that trace. To propagate the starting PC of the trace, we use a cyclic segmented parallel-prefix circuit with associative operator $a \otimes b = a$. To compute the offset into the trace, we use a cyclic segmented parallel-prefix circuit with associative operator $a \otimes b = a + b$. The addition inside the parallel-prefix nodes is performed by carry-save adders in order to keep the total gate delay down to $O(\log n)$. The execution stations holding the oldest instruction or the beginning address of a trace raise their segment bits and supply their trace offset. All other execution stations supply the constant 1.

In addition to its trace address, an execution station may also need to know the PC of its instruction. This is the case when an execution station detects a mispredicted branch and must generate the initial PC of a new trace. We can store the PC of every instruction within a trace and hand the PC to the execution station together with the instruction. Alternately, we can compute the PC of every execution station's instruction using another parallel prefix tree, just as we did for the trace offset. The only difference is the input to the tree. An execution station executing a relative branch that is predicted taken will send in the offset and a false segment bit. An execution station executing an absolute branch will send in the target address, once known, and a true segment bit.

Traces can be written into the instruction cache by the memory or the execution stations. If the instruction cache misses, then the trace must be created serially by fetching a predicted path from an instruction cache, much like in today's superscalars. The execution stations can also generate traces, however. Every time a trace fails to run to completion, a new trace is written into the instruction cache starting with the mispredicted branch. Each execution station writes its instruction into the trace, once its instruction commits. The starting PC of the new trace and the instruction's offset within the trace are computed by yet another prefix tree. This provides a small amount of parallelism for creating trace entries in the cache.

The trace caches in the literature, as well as ours, suffer from several problems. Both branch address caches and trace caches refill serially. We do not know how quickly our parallel trace cache refills in practice. Another concern is the amount of redundant data stored in the cache. Trace caches can keep exponentially many copies of particular instructions in the worst case. We have designed, and plan to describe elsewhere, a "pointer-jumping trace cache" that can quickly compute a trace while storing each instruction only a logarithmic number of times in the worst case.

4 Layout

We have so far concentrated on gate delays to understand the performance of the Ultrascalar. To accurately model critical-path delay we must not only consider the number of traversed gates, but also the length of traversed wires. The overall chip area is another important complexity measure as well. The critical-path delay and the area depend on a particular layout. In this section we show the Ultrascalar processor's layout using H-tree layouts for the datapath and for a fat-tree network that accesses an interleaved memory system. We compute the area and the lengths of the longest wires from that layout.

To lay out the Ultrascalar, we observe that all of the Ultrascalar interconnections consist entirely of cyclic segmented parallel prefixes connecting together the execution stations, plus fat-tree networks connecting the execution stations to memory. Both parallel-prefix circuits and fat-tree networks can be layed out using an H-tree layout. Figure 7 shows the floorplan of an Ultrascalar processor consisting of sixteen execution stations connected to interleaved on-chip caches via fully-fattened fat-trees (which are isomorphic to butterfly networks.) The nodes of the prefix trees are marked with DP. The stages of the butterfly are marked with MP (with the nodes of the butterfly inside the MP box.) Whereas the number of wires between any two DP nodes is constant, the number of nodes between two MP stages doubles at each level of the tree. The layout is called an H-tree layout because it consists of recursive H-shaped structures. Note that the four quadrants of the layout forms the tips of the letter "H", with the switches forming the intersections of the horizontal and vertical lines of the "H" and the internode connections forming the lines of the "H". Recursively, the four quadrants form another H-tree.[8]

[7] In fact, we can use the same network and memory modules as we used earlier.

[8] See [21, Section 3.1] for an introduction to layout of H-trees. See [9] for the layout of butterflies and fat-trees.

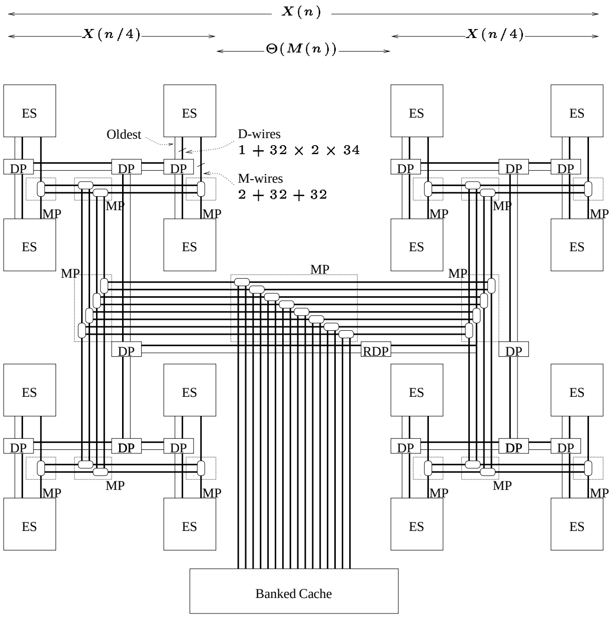


Figure 7: The floor plan for the VLSI layout of the Ultrascalar microprocessor with memory bandwidth linear in the window size, and a a simple serializing memory dependency checker. This version provides $O(I)$ memory bandwidth through the MP modules.

To demonstrate the scaling properties of the Ultrascalar, we are currently designing the processor in VLSI using the Magic design tool [10]. We have a complete processor core corresponding to the DP and ES modules in Figure 7. Our processor core executes a simple RISC instruction set architecture without floating point instructions. Figure 8 shows the plot of the second and third metal layers of a 64-station processor core.[9] To speed up our design time, we designed the datapath using CMOS standard cells. We did not worry about the optimal size of our gates or the thickness of our wires since these factors, once optimized, will remain constant for any size implementation. Because of the regularity of the Ultrascalar datapath, the design has so far taken less than four man months for a graduate student, including the time to learn the tools.

Area

The datapath's area is determined by the layout, as shown in Figure 7. We can compute the area of the circuit by observing that it is a recursive structure. To determine the area, we first determine the size of the bounding box for an n-wide Ultrascalar. As can be seen at the top of Figure 7, the width $X(n)$ of an n-wide Ultrascalar is equal to twice the width of an $n/4$-wide Ultrascalar plus the width of the wires. If we we provide bandwidth $M(n)$ memory operations per clock cycle to a subtree of size n then there are $\Theta(M(n))$ wires. (The wires for the datapath and other bookkeeping are only $O(1)$.) Thus we have the following recurence:

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

This recurence has solution[10]

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

[9] We obtained the plot of Figure 8 by taking a snapshot of a Magic X Window. Unlike our design, our plotting tools do not scale. Even a 16-station plot overflows virtual memory. We are rewriting the tool and hope to have a good 64-station plot by the time of the publication.

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

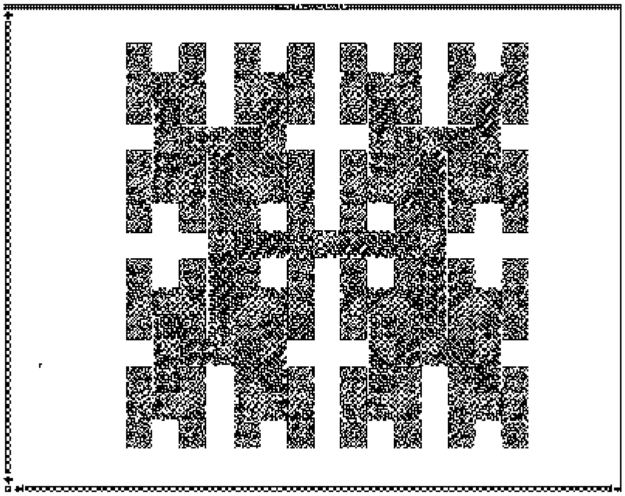


Figure 8: The VLSI layout of a 64-station Ultrascalar datapath corresponding to the DP and ES modules in Figure 7.

Thus, the area is

$$A(n) = (X(n))^2 = \begin{cases} \Theta(n) & \text{for Case 1,} \\ \Theta(n \log^2 n) & \text{for Case 2, and} \\ \Theta((M(n))^2) & \text{for Case 3.} \end{cases}$$

These bounds are optimal for a two-dimensional VLSI technology. In Case 1 the issue width is n, so the chip must hold at least n instructions, and thus the area must be $\Omega(n)$. In Case 2 the area must be $\Omega(n)$ and we have added at worst a $\log^2 n$ blowup.[11] In Case 3, the memory bandwidth requires that the edge of the chip be at least $\Omega(M(n))$ in order to get the memory bits in and out, giving an area of $\Omega((M(n))^2)$. For a three-dimensional technology, there are analogous layouts with optimal bounds.

Wire Length

Given the size of the bounding box for an n-wide Ultrascalar, we can compute the longest wire length 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 $L(n)$ be the wire length from the root to the leaves of an n-wide Ultrascalar. The wire length is the sum of

- the distance from the edge of the Ultrascalar to its internal switch (distance $X(n/4)$), plus
- the distance through the switch (the switch is $\Theta(M(n))$ on a side), plus
- the distance from the root of an $n/2$-wide Ultrascalar to its leaves (distance $L(n/2)$).

Thus we have the following recurence for $L(n)$:

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

In all three cass, this recurence has solution

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

That is, the wire lengths are the same as the side lengths of the chip to within a constant factor. 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 $2L(n)$. The memory signals only go up the tree so the longest memory signal is $L(n)$. The same optimality arguments that applied to area above apply to wire length here.[12]

[11] For Case 2 the bounds are nearly optimal. We will ignore Case 2's slight suboptimality for the rest of this paper.

[12] We assume that any processor must have a path from one end of the processor to the other.

Note that the switch above a subtree of n execution stations has area at least $O(M(n)^2)$ and so has plenty of area to implement all the switching and computation performed by the network. (For example, a merging network on n values can be embedded in area $\Theta(n^2)$ [20] with gate delay $\Theta(\log n)$ [2]. Such a sorting network can perform the hard work of the memory disambiguation of Section 3 with an overall gate delay of $\Theta(\log^2 n)$.)

The wire length depends primarily on how much memory bandwidth is needed. If we can reduce the required memory bandwidth, e.g. by using a cache in every execution station running a distributed coherency protocol, then we can reduce the area and the wire length of the processor. A brute-force design would provide memory bandwidth of $\Theta(n)$ for every n instructions, but it is reasonable to think that the required memory bandwidth for a sequence of n instructions may only be $O(n^{1/2})$, reducing the wire lengths from $\Theta(n)$ to $\Theta(n^{1/2}\log n)$. This asymptotic reduction in VLSI chip area underscores the importance of effective caching. We plan to study, in the future, how to build effective caches for Ultrascalar processors.

Critical Path Delay

Having analyzed the Ultrascalar's layout, it is now easy to see how its critical-path delays grow with the number of execution stations.[13] This is because the delay along any path in our implementation is simply linear in the number of gates plus the length of wires along that path. To achieve this linearity, we limit the fan-in and fan-out of each gate and insert repeater gates at constant intervals along wires. Since we stick exclusively to gates with a small, constant fan-in (the number of input wires) and fan-out (the number of output wires), each gate in our design drives a constant number of gate capacitances with a constant gate resistance. By breaking long wires into constant size segments connected by repeater gates, we make the capacitance and resistance of each wire segment also constant. Wire delays, including repeater delays, effectively become some constant fraction of the speed of light. Since the delay of each VLSI component is proportional to its resistance times its capacitance and since the resistances and capacitances of our gates and wire segments do not change with the size of our designs, the total delay along any path grows linearly with the number of gates and wire segments (i.e. wire length) traversed along that path. Specifically, the Ultrascalar's critical path delays due to gates grow logarithmically with the number of execution stations and its critical path delays due to wires grow at most linearly with the number of execution stations, giving

$$\tau_u = O(\log n + L(n)),$$

which is optimal.

5 Practical Performance Issues

Although the Ultrascalar has excellent scaling properties, much work remains to make the Ultrascalar practical. In this paper, we have presented a simple microarchitecture that is easy to analyze. We have shown that the microarchitecture scales well, but we have not optimized our microarchitecture for practical window sizes or issue widths. In fact, the Ultrascalar as we have presented it so far, cannot compete with today's superscalars using today's technology. The microarchitecture suffers from performance weaknesses brought about mostly by its inefficient passing of the entire register file to every instruction. In this section, we analyze the performance implications of this inefficiency and outline microarchitectures that avoid Ultrascalar's inefficiencies while still maintaining Ultrascalar's scaling properties.

The absolute area and critical path wire delays for a moderate size Ultrascalar datapath are infeasible even in today's best technology. Figure 9 lists area and critical path wire delays for different size Ultrascalars. We computed these data by scaling our layout to a 0.35 micron technology. We assumed a signal velocity of 17.5mm/ns achieved with optimal repeater placement as described in in [3].[14]

The Ultrascalar's large wire delays and area stem mostly from its wide datapath. Although a typical RISC instruction only reads two registers and writes one, The Ultrascalar passes the entire register file to and from every instruction. Passing the entire register file does not compromise the Ultrascalar's scaling properties since a register file contains a constant number of registers, but it does introduce large constants hiding in the Θ.

One way to reduce the constants is by combining the best properties of the superscalar processor with the Ultrascalar. Although, so far, we have described each execution station as holding a single instruction, there is no reason

[13] The number of execution stations is the same as the issue width, the fetch width, and the instruction window size of the Ultrascalar processor.

[14] Our wire delay estimates are pessimistic by a small constant factor because the Ultrascalar datapath is layed out in the metal-3 and metal-2 layers, whereas the delay calculations in [3] are for the metal-1 layer.

Number of Execution Stations	Area	Critical-Path Wire Delay
64	10.5cm × 7.6cm	12.4 ns
32	8.1cm × 4.9cm	8.9 ns
16	4.2cm × 3.8cm	5.8 ns
4	1.6cm × 1.4cm	1.9 ns

Figure 9: Area and critical path wire delays for different size Ultrascalars in a 0.35 micron technology. (This layout provides for 1 memory operation per clock cycle.) In TI's proposed 0.07 micron technology [19] the wire lengths and delays would presumably be about reduced by about a factor of 5.

why an execution station cannot hold and execute a sequence of instructions instead. The sequence of instructions within an execution station can be executed using, for example, a traditional superscalar processor core. Since, for small enough window size, a superscalar processor has smaller critical path delay and smaller area than an Ultrascalar processor, placing superscalar processor cores at the leaves of the Ultrascalar datapath can shorten the overall critical path delay and reduce area. At the same time, the Ultrascalar datapath can provide time- and area-efficient routing among the many superscalar cores.

The hybrid microarchitecture just described bears resemblence to clustering [4, 11, 5]. Clustering avoids some of the circuit delays associated with large superscalar instruction windows by assigning ALUs and outstanding instructions to one of several smaller clusters with a smaller window. Instructions that execute in the same cluster can communicate quickly, but when dependent instructions execute in different clusters, an extra clock delay is introduced to resolve the dependency. The reported schemes are limited to two clusters. Decoded and renamed instructions are assigned to one of the two clusters using simple heuristics. It is not clear how well these heuristics will scale to large numbers of clusters. In addition, other slow components of the superscalar processor, such as the renaming logic, are not addressed by clustering. Like clustering, our hybrid microarchitecture also separates outstanding instructions into clusters. The heuristic used to assign instructions to clusters is their proximity within the dynamic instruction sequence. Thus, instructions that are near each other in the serial execution order are likely to be in the same superscalar core or the same subtree. Instructions that are far from each other communicate via the Ultrascalar's logarithmic depth network. The Ultrascalar placement heuristic is probably not as good as the clustering heuristic when there are two clusters or two execution stations. It is not clear how to build larger clusters, however, whereas the hybrid Ultrascalar does scale up.

The second enhancement that can reduce the Ultrascalar's wide datapath is tagging. Fundamentally, there is no reason why the Ultrascalar needs to pass the entire register file to and from every execution station. Much like the counterflow pipeline [18], the Ultrascalar datapath can pass each execution station only two arguments tagged with their register numbers and accept one result tagged with its register number. The Ultrascalar datapath can merge these incremental register file updates as they propagate through the tree. The resulting tree can be layed out as a tree that fattens as it ascends towards the root for the lower $\lg L$ levels of the tree, where L is the number of logical registers. We plan to describe this mechanism elsewhere.

An additional way to mitigate the wire delays introduced by the Ultrascalar's wide datapath, is through the design's timing discipline. For example, we can improve the average-case routing delay by pipelining the datapath network of Figure 3. We can separate every subtree containing k execution stations from the rest of the datapath network by registers. The instructions within a subtree can then route in the same clock cycle in which they compute. Instructions in two separate subtrees communicate using an additional clock cycle. Since in a tree, most connections are local, the slowdown may not be very great in the typical case.[15] The Ultrascalar also appears lends itself well to an asynchronous self-timed logic methodology.

Aside from the Ultrascalar's wide datapath, another issue of practical concern is the relatively large number of ALUs. The Ultrascalar assigns an ALU to every instruction. We believe that the large number of ALUs will not be a problem in the future, because in a billion-transistor chip, a full ALU will probably require only about 0.1% of the chip area.[16] Even if ALUs become a serious bottleneck, the ALUs can be shared. A cyclic segmented parallel-prefix tree can schedule an ALU among m execution stations with only $\Theta(\log m)$ gate delays, and with relatively small area.[17]

[15] Note that the CM-5 control network uses a pipelined parallel-prefix tree [8].

[16] Our standard-cell ALU uses only about 13,000 transistors, but it includes no floating point support.

[17] Palacharla et al [11] also show how to schedule ALUs with only $\Theta(\log m)$ gate delays. But their scheduler does not wrap around. It statically

6 Summary

Other recent work has addressed the scalability of superscalar processor microarchitecture. In Section 3, we outlined current research on improving the effective instruction bandwidth that could benefit the Ultrascalar as well as traditional superscalars. In Section 5, we compared the Ultrascalar to existing and proposed clustering schemes that cluster instructions around smaller instruction windows, but do not address the scalability of broadcasting or renaming. In Section 2, we pointed out the counterflow which deeply pipelines long chains of instructions, but refills instructions serially and routes using a linear depth rather than logarithmic depth network. Several additional approaches to scaling today's uniprocessors rely on course-grained instruction-level parallelism. MultiScalar processors [17] take course-grained threads that must be executed serially, and speculatively execute them, noticing after the fact when a data dependency has been violated. MultiScalar processors do not address the problem of exploiting fine-grained ILP. DataScalar processsors [1] execute instructions redundantly, and require an efficient broadcast mechanism, which appears to limit its scalability.

One often-heard slogan states that "the computer is the network" (or *vice versa.*) To scale to greater and greater issue widths the processor too must become a network. We have presented a processor implementation that dramatically differs from today's designs, scaling efficiently, and with low design effort, to large issue widths by converting the processor's datapath into several logarithmic-depth networks. Although the Ultrascalar's implementation differs dramatically from today's processors, the Ultrascalar exploits the same parallelism as more traditional processors, by providing out-of-order issue, register renaming, and speculation. To exploit even more parallelism, we have proposed improvements to the data and instruction memories. Several improvements to the Ultrascalar datapath are possible to reduce the constants.

Although much work remains in making the Ultrascalar practical, we envision the Ultrascalar circuits improving processor performance in the near term as well as the distant future. For example in the short term many of the parallel prefix circuits introduced in Section 2 can efficiently compute dynamic properties of instructions for a traditional superscalar processor. For instance, a cyclic, segmented parallel prefix can determine which instructions have committed, select the oldest instruction, or assign instructions to available resources. In the medium term, a hybrid superscalar-Ultrascalar design (outlined in Section 5) can extend the clustering concept to four or eight clusters. In the longer term, if instruction-level parallelism allows, an optimized Ultrascalar processor can harvest instruction-level parallelism from sequences of hundreds or even thousands of instructions. With its attractive scaling properties, the Ultrascalar highlights a promising direction for the next decade of processors.

Finally, the Ultrascalar processor demonstrates that VLSI algorithms and analysis are an important tool for processor architects.

References

[1] Doug Burger, Stefanos Kaxiras, and James R. Goodman. DataScalar architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, Denver, Colorado, 2–4 June 1997. ACM SIGARCH and IEEE Computer Society TCCA. `ftp://ftp.cs.wisc.edu/galileo/papers/ISCA97_ds.ps``ftp://ftp.cs.wisc.edu/galileo/papers/ISCA97_ds.ps`.

[2] 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.

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

[4] 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.

[5] 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.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1990.

[7] 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.

assigns the highest priority to the left-most request. It also uses relatively larger area than is required by a parallel prefix tree.

[8] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996. `ftp://theory.lcs.mit.edu/pub/bradley/jpdc96.ps.Z``ftp://theory.lcs.mit.edu/pub/bradley/jpdc96.ps.Z`.

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

[10] 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.

[11] 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``http://www.ece.wisc.edu/~jes/papers/isca.ss.ps`. See also [12].

[12] 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``ftp://ftp.cs.wisc.edu/sohi/complexity.report.ps.Z`.

[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``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``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. IEEE Computer Society TC-MICRO and ACM SIGMICRO. `http://www.cs.wisc.edu/~ericro/TC_micro29.ps``http://www.cs.wisc.edu/~ericro/TC_micro29.ps`.

[16] Richard M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[17] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 414–425, Santa Margherita Ligure, Italy, 22–24 June 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News*, 23(2), May 1994.

[18] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.

[19] TI prepares to build chips based on smallest announced transistors. `http://www.ti.com/corp/docs/pressrel/1998/c98048.htm``http://www.ti.com/corp/docs/pressrel/1998/c98048.htm`, 26 August 1998.

[20] Clark D. Thompson. The VLSI complexity of sorting. *IEEE Transactions on Computers*, C-32:1171–1184, December 1983.

[21] J. D. Ullman. *Computational Aspects of VLSI*. Principles of Computer Science Series. Computer Science Press, 1984.

[22] 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.