THE RAW MICROPROCESSOR: A COMPUTATIONAL FABRIC FOR SOFTWARE CIRCUITS AND GENERAL-PURPOSE PROGRAMS

WIRE DELAY IS EMERGING AS THE NATURAL LIMITER TO MICROPROCESSOR SCALABILITY. A NEW ARCHITECTURAL APPROACH COULD SOLVE THIS PROBLEM, AS WELL AS DELIVER UNPRECEDENTED PERFORMANCE, ENERGY EFFICIENCY, AND COST EFFECTIVENESS.

Michael Bedford Taylor
Jason Kim
Jason Miller
David Wentzlaff
Fae Ghodrat
Ben Greenwald
Henry Hoffman
Paul Johnson
Jae-Wook Lee
Walter Lee
Albert Ma
Arvind Saraf
Mark Seneski
Nathan Shnidman
Volker Strumpen
Matt Frank
Saman Amarasinghe
Anant Agarwal
Massachusetts Institute of Technology

The Raw microprocessor consumes 122 million transistors; executes 16 different load, store, integer, or floating-point instructions every cycle; controls 25 Gbytes/s of input/output (I/O) bandwidth; and has 2 Mbytes of on-chip distributed L1 static RAM providing on-chip memory bandwidth of 57 Gbytes/s. Is this the latest billion-dollar, 3,000 man-year processor effort? In fact, it took only a handful of graduate students at the Laboratory for Computer Science at MIT to design and implement Raw.

Our research addresses a key technological problem for microprocessor architects: How to leverage growing quantities of chip resources even as wire delays become substantial.

The Raw research prototype uses a scalable instruction set architecture (ISA) to attack the emerging wire-delay problem by providing a parallel, software interface to the gate, wire, and pin resources of the chip. An architecture that has direct, first-class analogs to all of these physical resources will ultimately let programmers achieve the maximum amount of performance and energy efficiency in the face of wire delay. Existing architectural abstractions, such as interrupts, caches, context switches, and virtualization can continue to be supported in this environment, even as a new low-latency communication mechanism—the static network—enables new application domains.

Technology trends

Until recently, the abstraction of a wire as an instantaneous connection between transistors has shaped assumptions and architectural designs. In an interesting twist, just as the clock frequency of processors has risen exponentially over the years, the fraction of the chip that is reachable by a signal in a single clock cycle has decreased exponentially.[1] Thus, the idealized wire abstraction is becoming less and less representative of reality. Architects now need to explicitly account for wire delay in their designs.

Today, it takes on the order of two clock cycles for a signal to travel from edge-to-edge (roughly fifteen mm) of a 2-GHz processor

0272-1732/02/$17.00 © 2002 IEEE

An evolutionary response for current instruction set architectures

Figure A shows how we believe today's microarchitectures will adapt as effective silicon area and pin resources increase and as wire delay worsens. Designers will want to utilize the increased silicon resources, while maintaining high clock rates. We envision a high-frequency execution core, containing several nearby clustered ALUs, with speculative control guessing to which ALU cluster to issue. Around this core is a host of pipelined floating-point units (which are less latency sensitive), prefetch engines, multilevel cache hierarchies, speculative control, and other out-of-band logic that's focused on making the tiny execution core run as efficiently and as fast as possible. In addition, since most conventional instruction set architectures (ISAs) do not have an architectural analog to pins, most pins will be allocated to wide, external-memory interfaces that are opaque to the software.

The ISA gap between software-usable processing resources and the actual amount of underlying physical resources is going to steadily increase in these future designs. Even today, it is easy to tell that the percentage of silicon performing actual computation has dropped quadratically over time. The Compaq Alpha 21464 design—an eight-way issue superscalar—is in excess of 27 times as large as the original two-way issue 21064. The area of the management logic dwarfs the area occupied by the ALUs, and system performance is getting increasingly nondeterministic and sensitive to the particulars of the program implementation. Intel, for instance, has produced hundreds of pages that suggest methods of improving system performance by avoiding stall conditions in the Pentium 4 microarchitecture. Furthermore, the power consumption, as well as design and verification costs, of these increasingly complex architectures is skyrocketing.

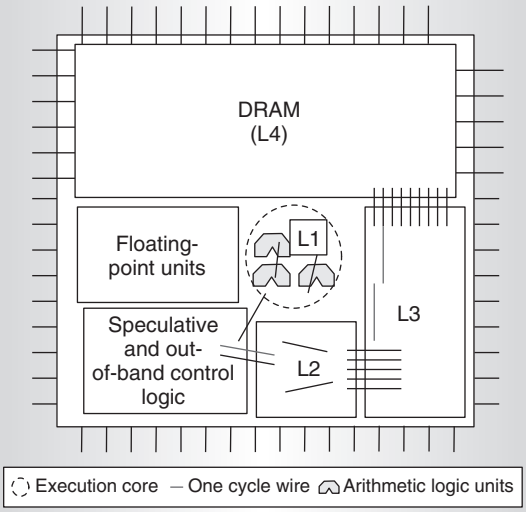

Figure A. How today's microarchitectures might adapt to worsening wire delay as effective silicon area and pin resources increase.

die. Processor manufacturers have strived to maintain high clock rates in spite of the increased impact of wire delay. Their innovation has been at many levels. The transition from aluminum to copper wires has reduced the resistance and thus the resistance-capacitance delay of the wires. Process engineers have also altered the aspect ratio of the wires to reduce resistance. Finally, the introduction of low-k dielectrics will provide a one-time improvement in wire delay.

Unfortunately, materials and process changes have not been sufficient to solve the problem. Forced to worry about wire delays, designers place the logic elements of their processors very carefully, minimizing the distance between communicating transistors on critical paths. In the past, precious silicon area dictated logic reuse, but designers now freely duplicate logic to reduce wire lengths—for example, the adders in the load/store unit and in the arithmetic logic unit (ALU).

Even microarchitects are making concessions to wire delay. The architects of Compaq's Alpha 21264 were forced to split the integer unit into two physically dispersed clusters, with a one-cycle penalty for communication of results between clusters. More recently, the Intel Pentium 4 architects had to allocate two pipeline stages solely for the traversal of long wires.

The wire delay problem will only get worse. In the arena of 10-GHz processors, designers will experience latencies of 10 cycles or more across a processor die. It will become increasingly more challenging for existing architectures to turn chip resources into higher performance. See the "An evolutionary response for current instruction set architectures" sidebar for how we think existing architectures will attempt to overcome this challenge.

The Raw microprocessor

We designed Raw[2] to use a scalable ISA that provides a parallel software interface to the gate, wire, and pin resources of a chip. An architecture with direct, first-class analogs to all of these physical resources lets programmers extract the maximum amount of performance and energy efficiency in the face of wire delay. In effect, we try to minimize the ISA gap by exposing the underlying physical resources as architectural entities.

The Raw processor design divides the usable silicon area into 16 identical, programmable tiles. Each tile contains

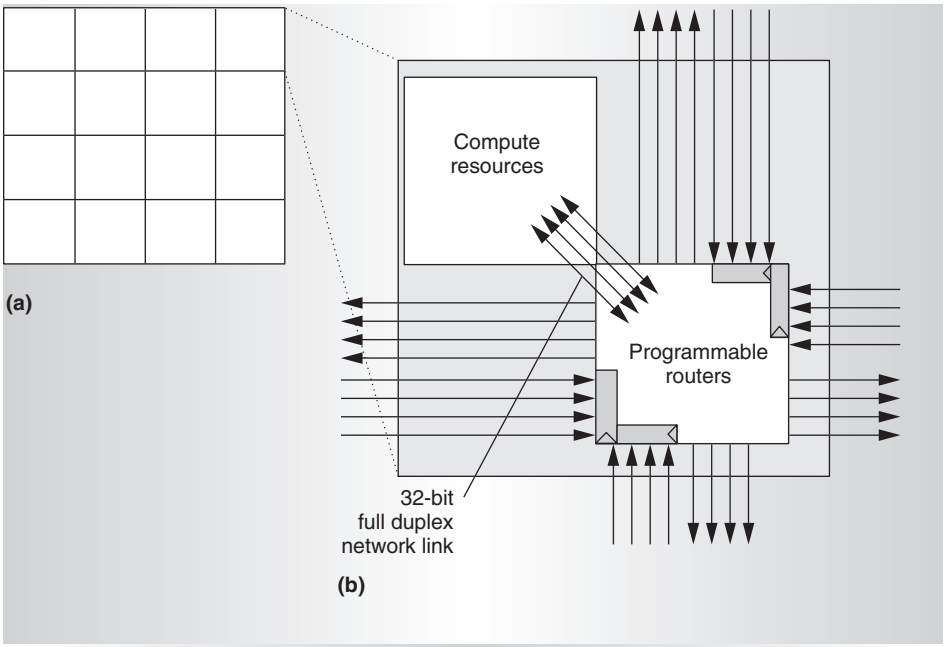

Figure 1. On-chip interconnects in Raw. The Raw microprocessor comprises 16 tiles (a). Each tile (b) has computational resources and four networks, each with eight point-to-point 32-bit buses to neighbor tiles.

- one static communication router;
- two dynamic communication routers;
- an eight-stage, in-order, single-issue, MIPS-style processor;
- a four-stage, pipelined, floating-point unit;
- a 32-Kbyte data cache; and
- 96 Kbytes of software-managed instruction cache.

We sized each tile so that the time for a signal to travel through a small amount of logic and across the tile is one clock cycle. Future Raw processors will have hundreds or perhaps thousands of tiles.

The tiles interconnect using four 32-bit full-duplex on-chip networks, consisting of over 12,500 wires, as Figure 1 shows. Two networks are static (routes specified at compile time) and two are dynamic (routes specified at runtime). Each tile only connects to its four neighbors. Every wire is registered at the input to its destination tile. This means that the length of the longest wire in the system is no greater than the length or width of a tile. This property ensures high clock speeds, and the continued scalability of the architecture.

The Raw ISA exposes these on-chip networks to the software, enabling the programmer or compiler to directly program the wiring resources of the processor and to carefully orchestrate the transfer of data values between the computational portions of the tiles—much like the routing in a full-custom application specific integrated circuit (ASIC). Effectively, the wire delay manifests itself to the user as network hops. It takes six hops for a data value to travel from corner to corner of the processor, corresponding to approximately six cycles of wire delay.

On the edges of the network, the network buses are multiplexed in hardware down onto the pins of the chip using logical channels,[3] as Figure 2 (next page) shows. To toggle a pin, the user programs the on-chip network to route a data value off the side of the array. Our 1,657 pin, ceramic-column grid-array package provides 14 full-duplex, 32-bit, 7.5-Gbps I/O ports at 225 MHz, for a total of 25 Gbytes/s of bandwidth. The design does not require this many pins; rather, we use this number to illustrate that no matter how many pins a package has (100 or 10,000), Raw's scalable architectural mechanism lets the programmer put them to good use. Fewer pins merely require more multiplexing. Alter-

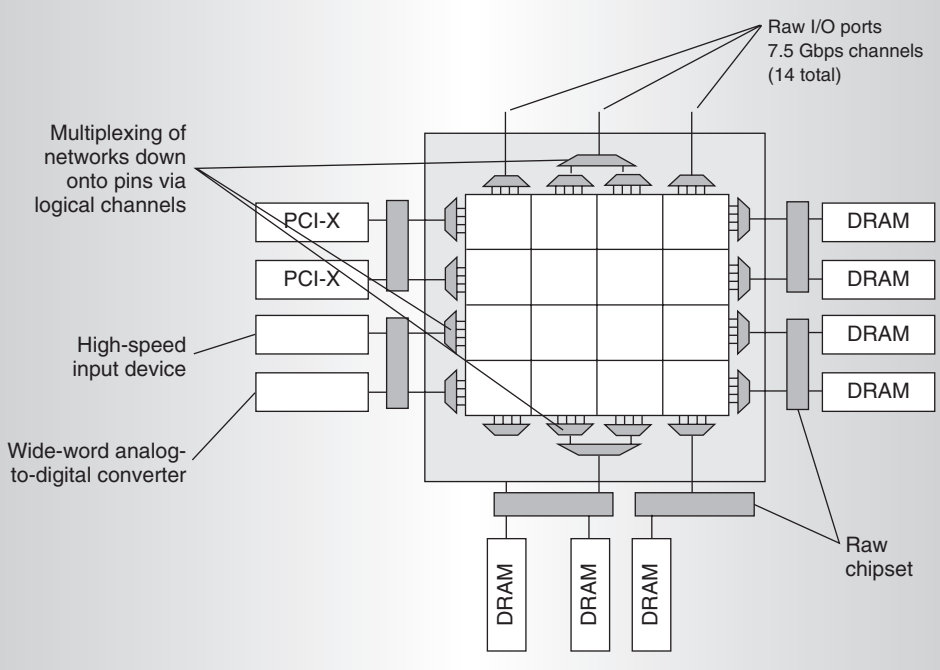

Figure 2: Pin multiplexing and device usage in Raw.

Table 1. How Raw converts physical resources into architectural entities.*

Physical entity	Raw ISA analog	Conventional ISA analog
Gates	Tiles, new instructions	New instructions
Wire delay	Network hops	None
Pins	I/O ports	None

*Conventional ISAs attempt to utilize increasing gate quantities through the addition of new instructions (like parallel SIMD instructions) and through dynamic mapping of operations to a small number of architecturally invisible ALUs. Wire delay is typically hidden through pipelining and speculation, and is reflected to the user in the form of dynamic stalls for non-fast-path and mispredicted code. Pin bandwidth is hidden behind speculative cache-miss hardware prefetching and large line sizes.

nately, a small pin-count package can be supported by bonding out only a subset of the ports.

The Raw I/O port is a high-speed, simple (a three-way multiplexed I/O port has 32 data and five control pins for each direction), and flexible word-oriented abstraction that lets system designers proportion the quantities of I/O devices according to the application domain's needs. Memory intensive domains can have up to 14 dedicated interfaces to DRAM. Other applications may not have external memory—a single ROM hooked up to any I/O port is sufficient to boot Raw so that it can execute out of the on-chip memory. In addition to transferring data directly to the tiles, off-chip devices connected to Raw I/O ports can route through the on-chip networks to other devices to perform direct memory accesses (DMAs). We plan to hook up arrays of high-speed data input devices, including wide-word analog-to-digital converters, to experiment with Raw in domains that are I/O, communication, and compute intensive.

Architectural entities

The conventional ISA has enjoyed enormous success because it hides the details of the underlying implementation behind a well-defined compatibility layer that matches the underlying implementation substrate fairly well. Much as the existence of a physical multiplier in a processor merits the addition of a corresponding architectural entity (the multiply instruction), the prominence of gate resources, wire delay, and pins will soon merit the addition of corresponding architectural entities. Furthermore, we expose these entities in a way that will allow subsequent generations of Raw processors to be backward compatible.

Table 1 contrasts how the Raw ISA and conventional ISAs expose gates, wire delay, and pins to the programmer. Because the Raw ISA has interfaces that are more direct, Raw processors will have more functional units, as well as more flexible and more efficient pin utilization. High-end Raw processors are likely to have more pins, because the architecture is better at turning pin count into performance and functionality. Finally, Raw processors will be more predictable and have higher clock frequencies because of the explicit exposure of wire delay.

This exposure makes Raw scalable. Creating subsequent, more powerful, generations of the processor is straightforward; we simply stamp out as many tiles and I/O ports as the

silicon die and package allow. The design has no centralized resources, global buses, or structures that get larger as the tile or pin count increases. Finally, wire length, design complexity, and verification complexity are all independent of transistor count.

Application domains

The Raw microprocessor runs computations that form a superset of those run on today's general-purpose processors. Our goal is to run not just conventional scalar codes (for example, Specint and Specfp), but word-level computations that require so much performance that they have been consigned to custom ASIC implementations. If an application can take advantage of the customized placement and routing, the ample gates, and the programmable pin resources available in an ASIC process, it should also benefit from the architectural versions of those same resources in the Raw microprocessor. For instance, our first-cut, untuned implementation of a software Gigabit Internet protocol router on a 225-MHz, 16-tile Raw processor runs more than fives times faster than a hand-tuned implementation on a 700-MHz Pentium III processor. Additionally, an implementation of video median filter on 128 tiles attained a 57-time speedup over a single Raw tile.

Unlike an ASIC, however, applications for Raw can be written in a high-level language such as C or Java, or new languages such as StreamIt,[4] and the compilation process takes minutes, not months. We call applications that leverage the Raw static network's ASIC-like place-and-route facility *software circuits*. Sample software circuits we are experimenting with include gigabit routers, video and audio processing, filters and modems, I/O protocols (RAID, SCSI, FireWire) and communications protocols (for cellular phones, multiple channel cellular base stations, high-definition TV, Bluetooth, and IEEE 802.11a and b). These protocols could run as dedicated embedded hosts, or as a process on a general-purpose machine.

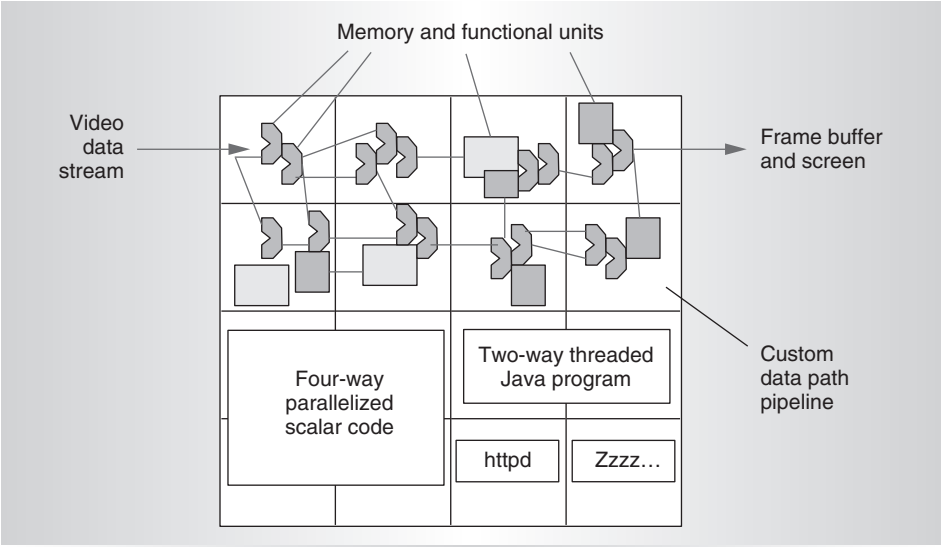

Figure 3. Application mapping onto a Raw microprocessor.

Application mapping

The Raw operating system allows both space and time multiplexing of processes. Thus, not only can a Raw processor run multiple independent processes simultaneously, it can context switch them in and out as on a conventional processor. The operating system allocates a rectangular-shaped number of tiles (corresponding to physical threads that can themselves be virtualized) proportional to the amount of computation that is required by that process. When the operating system context-switches in a given process, it finds a contiguous region of tiles that corresponds to the dimension of the process, and resumes the execution of the physical threads. We employ this gang scheduling policy because the physical threads of the process are likely to communicate with each other. Continuous or real-time applications can be locked down and will not be context-switched out.

Figure 3 shows a Raw processor running multiple processes simultaneously. Traditional applications, including threaded Java programs, message passing interface (MPI) codes, and server applications, use Raw as a high-density multiprocessor. The corner tile is in sleep mode to save power. The four-tile block is running an automatically parallelized[5] scalar code. The top eight tiles in Figure 3 illustrate the more novel software circuit usage of the tiles. Video data is streamed in over the pins, transformed by a filtering operation, and then streamed out to a frame buffer and screen. In

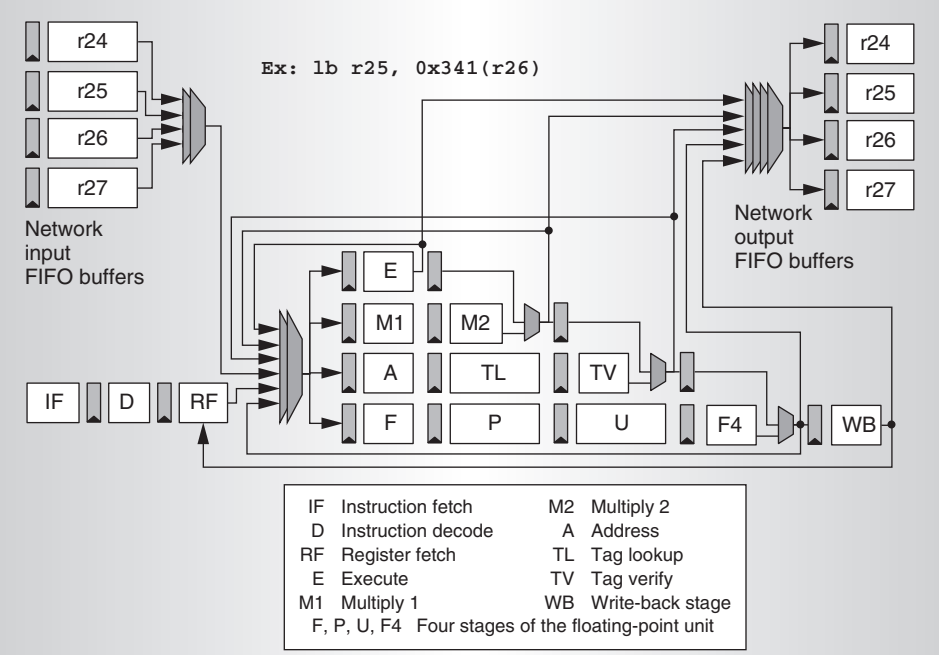

Figure 4. Raw compute processor pipeline.

this situation, the tiles work together, parallelizing the computation. We assign operations to tiles in a manner that minimizes congestion and then configure the network routes between these operations. This is very much like the process of designing a customized hardware circuit. In Raw, the compiler performs this customization.[4,5] This customization can also be done manually using Raw's assembly code.

To make both parallelized scalar codes and software circuits work, we need a very low-latency network—the faster the network, the greater the range of applications that can be parallelized. Multiprocessor networks designed for MPI programs have latencies on the order of 1,000 cycles; state-of-the-art networks have latencies of 30 cycles.[6] The Raw network can route the output of the ALU on one tile to the input of the ALU of a neighboring tile in three cycles.

Design decisions

The Raw tile design crystallized around the need to provide low-latency communication for efficient execution of software circuits and parallel, scalar codes. At the same time, we wanted to provide scalable versions of the standard toolbox of useful architectural constructs, such as data and instruction virtualization, caching, interrupts, context switches, address spaces, latency tolerance, and event counting. To achieve these goals, a Raw tile employs its compute processor, static router, and dynamic routers. The static router manages the two static networks, which provide the low-latency communication required for software circuits and other applications with compile-time predictable communication. The dynamic routers manage the dynamic networks, which transport unpredictable operations like interrupts, cache misses, and compile-time unpredictable communication (for example, messages) between tiles.

Compute processor

Our focus in designing the compute processor was to tightly integrate coupled network interfaces into the processor pipelines. We wanted to make the network first class in every sense to maximize its utility. The most common network interfaces are memory mapped; other networks use special instructions for sending and receiving.[6,7] The most aggressive processor networks are register mapped and do not require a special send or receive command; instructions can target the networks just as easily as registers.[3]

Our design takes network integration one step further: The networks are not only register mapped but also integrated directly into the bypass paths of the processor pipeline. This makes the network ports truly first-class citizens in the architecture. Figure 4 shows how this works. Registers 24 through 27 are mapped to the four on-chip physical networks. For example, a read from register 24 will pull an element from an input FIFO buffer, while a write to register 24 will send the data word out onto that network. If data is not available on an input FIFO buffer, or if an output FIFO buffer does not have enough room to hold a result, the processor will stall

in the register fetch stage. The instruction format also provides a single bit in the instruction, which allows the instruction to specify two output destinations: one network or register, and the network implied by register 24 (the first static network). This gives the tile the option of keeping local copies of transmitted values.

Each output FIFO buffer connects to each pipeline stage. The FIFO buffers pull the oldest value out of the pipeline as soon as it is ready, rather than just at the write-back stage or through the register file.[3] This decreases the latency of an ALU-to-network instruction by as much as four cycles for our eight-stage pipeline. This logic is exactly like the standard bypass logic of a processor pipeline except that it gives priority to older instructions rather than newer instructions.

In early processor designs, the register file was the central communication mechanism between functional units. Starting with the first pipelined processors, the bypass network became largely responsible for the communication of active values, and the register file evolved into a dumping ground or checkpointing facility for inactive values. The Raw networks (the static networks in particular) extend this trend and are in essence 2D bypass networks serving as bridges between the bypass networks of separate tiles. The low latency of in-order, intertile ALU-to-ALU operand delivery distinguishes Raw from previous systolic or message-passing systems.[3,6,7] The integration of networks into the pipelined bypass path of the compute processor reflects our view that scalar data transport among functional units on adjacent tiles is as important as that which occurs among functional units within a single tile. The resulting low latency of intertile communication allows the Raw static network to perform customized scalar data routing with ASIC-like latencies.

The early bypassing of values to the network has some challenging effects on pipeline operation. Perhaps most importantly, it changes the semantics of the compute processor's commit point. An instruction that has had a value bypassed out early has created a side effect, which makes it difficult to squash the instruction in a later stage. The simplest solution we have found for this is to place the commit point at the execute stage.

Static router

For software circuits and parallel scalar codes, we use the two static networks to route values between tiles. The static networks provide ordered, flow-controlled, and reliable transfer of single-word operands and data streams between the tiles' functional units. The operands need to be delivered in order so that the instructions issued by the tiles are operating on the correct data. Flow control of operands allows the program to remain correct in the face of unpredictable architectural events such as cache misses and interrupts.

The static router is a five-stage pipeline that controls two routing crossbars and thus two physical networks. Each crossbar routes values between seven entities—the static router pipeline; the north, east, south, and west neighbor tiles; the compute processor; and the other crossbar. The static router uses the same fetch unit design as the compute processor, except it fetches a 64-bit instruction word from the 8,096-entry instruction memory. This instruction simultaneously encodes a small command (conditional branches with or without decrement, and accesses to a small register file) and 13 routes (one for each unique crossbar output) for a total of 14 operations per cycle per tile.

For each word sent between tiles on the static network, there must exist a corresponding instruction in the instruction memory of each router on the word's path. These instructions are typically programmed at compile time and are cached just like the instructions of the compute processor. Thus, the static routers collectively reconfigure the entire communication pattern of the network on a cycle-by-cycle basis. Further, because the router program memory is large and also cached, there is no practical architectural limit on the number of simultaneous communication patterns supported in a computation. In this manner, the communication and compute operations are treated with equal importance. This mechanism distinguishes Raw's network from previous systems such as iWarp or Numesh.[3,8]

Because the static router knows what route will be performed long before the word arrives, route preparations can be pipelined. This allows data word routing immediately upon the word's arrival. This low latency is critical for exploiting instruction-level parallelism in scalar codes. Dynamic routers have

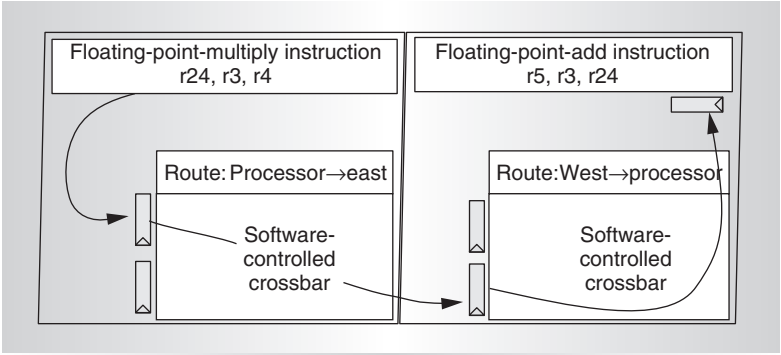

Figure 5. Two tiles communicating over the static network.

more latency than the static router (despite the same bandwidth) because they have to wait for a header word to arrive before they can initiate preparations for a route. Thus, dynamic networks are better suited for long data streams and not scalar transport.

The static router is flow-controlled, and it proceeds to the next instruction only after all of the routes in a particular instruction are completed. This ensures that destination tiles receive incoming words in a known order, even when tiles suffer branch mispredicts, cache misses, interrupts, or other unpredictable events.

The static router provides single-cycle-per-hop latencies and can route two values in each direction per cycle. Figure 5 shows two tiles communicating across the static network. The word computed by the floating-point multiply instruction spends one cycle in each static router and one cycle in the decode stage of the target tile. This gives a total latency of three cycles for a word to travel from the output of the ALU of one tile to the input of the ALU of the next tile. Unlike this example, actual Raw programs have more random and dense communication patterns, which reflect the internal flow of data between operations in the original high-level language source.

The dynamic networks

Early on in the Raw project, we realized the need to support dynamic as well as static events. We thus added a pair of dimension-ordered, wormhole-routed dynamic networks to the architecture.[9] To send a message on one of these networks, the user injects a single header word that specifies the destination tile (or I/O port), a user field, and the length of the message. The user then sends up to 31 data words. While this is happening, the message worms its way through the network to the destination tile. Our implementation of this network takes one cycle per hop, plus an extra cycle for every hop that turns. (We use an optimization that exploits the fact that most routes are straight.) On an uncongested network, the header reaches the destination in 2 + X + 1 + Y + 2 cycles. That is, two cycles of latency to leave the compute processor (this counts as a hop and a turn), a number of X (that is, horizontal) hops, one hop if a turn is required, a number of Y hops, and then a hop and a turn to enter the compute processor.

One major concern with dynamic networks is deadlock caused by the over commitment of buffering resources. Classically, there are two solutions for deadlock: avoidance and recovery. Deadlock avoidance requires users to limit their usage to a set of disciplines that have been proven not to deadlock. Unfortunately, this limits the network's usefulness. Deadlock recovery places no restrictions on network use, but requires that the network be drained to some source of copious memory when the network appears to be deadlocked. Unfortunately, if that out-of-band memory system itself uses deadlock recovery, the system could fail. Our solution uses a pair of otherwise identical networks: The memory network has a restricted usage model that uses deadlock avoidance, and the general network is unrestricted and uses deadlock recovery. If the general network deadlocks, an interrupt routine is activated that uses the memory network to recover.

Trusted clients—operating system, data cache, interrupts, hardware devices, DMA, and I/O—use the memory network. Each client is not allowed to block on a network send unless it can guarantee that its input buffers can sink all messages that it might be receiving. Alternatively, a client can guarantee that it will not block on a network send if it has enough private buffering resources on the path to its destination. Clients are each allocated a number of buffers to accomplish their basic functionality. For large, high-performance transfers, the clients can negotiate permission with the operating system for temporary use of a pool of additional buffers associated with each I/O port.

Untrusted clients use the general network and rely on the hardware deadlock recovery

mechanism to maintain forward progress when deadlock occurs. The operating system programs a configurable counter on the compute processor to detect if words have waited too long on the input.[6] This counter causes an interrupt so that the network can be drained into DRAM. A separate interrupt then allows the general network input port to be virtualized, substituting in the data from the DRAM.

Because it is a user-level construct, the general network is virtualized for each group of tiles that corresponds to a process. The upper left tile is considered Tile 0 for the purposes of this process. On a context switch, the contents of the general and static networks are saved off, and the process and its network data can be restored at any time to a new offset on the Raw grid.

Both the hardware caches and the software share the memory network. On a cache miss, the hardware cache consults a configurable hash function to map addresses to destination ports or tiles. The cache then issues a header word, like every other client of the network, and then a sequence of words that is interpreted by the DRAM controller. On a cache fill, the cache state machine will also wait for the result and then transfer words into the cache memory. Just like any other I/O device on the network, the DRAMs are equally accessible by hardware and software. It is often useful to write codes that operate on large data blocks that stream directly in and out of the DRAMs.

The Raw processor supports parallel implementation of external (device I/O) interrupts; each tile can process an interrupt independently of the others. The interrupt controller(s) (implemented by a dedicated tile or as part of the support chipset) signals an interrupt to a particular tile by sending a special one-word message through an I/O port. The tile's network hardware checks for that message and transparently pulls it off the network and sets the compute processor's external interrupt bit. When the compute processor services the interrupt, it queries the interrupt controller for the cause of the interrupt and then contacts the appropriate device or DRAM.

Implementation

We implemented the Raw chip 16-tile prototype in IBM's SA-27E, 0.15-micron, six-level, copper, ASIC process. Although the Raw array is only 16×16 mm, we used an 18.2×18.2-mm die to allow a high pin-count package. The 1,657 pin, ceramic-column grid-array package provides 1,080 high-speed transceiver logic I/O pins. We estimate that the chip consumes 25 W, mostly in memory accesses and pins. We quiesce unused functional units and memories and tristate unused data I/O pins. We targeted a 225-MHz worst-case frequency (average-case frequency is typically 25 percent higher), which is competitive with other 0.15-micron ASIC processors, like IRAM from the University of California, Berkeley, and the customizable processors from Tensilica.

We pipelined our processor aggressively and treated control paths very conservatively to avoid spending significant periods closing timing in the back end. Despite this, we found that wire delay inside the tile was large enough that placement could not be ignored as an issue. We created a library of routines (7,000 lines of code) that automatically places the majority of the structured logic in the tile and around the perimeter of the chip. This structured logic is visible in Figure 6, a screen capture of the Raw tile placement. Figure 7 shows a screen capture of placement for the entire Raw chip. The replicated 4×4-pattern in the middle is the array of Raw tiles. The semistructured logic around the edges is the I/O multiplexing logic. Figure 8 (next page) gives a detailed floorplan of the Raw tile.

The placement routines dropped cycle time from 8 to 4 ns, which matches the Synopsys synthesis tool's timing estimates. The synthesis, back-end processing, and our placement infrastructure can turn our Verilog source into a fully placed chip in approximately 6 hours on one machine. We used a logic emulator donated by IKOS Technologies coupled with

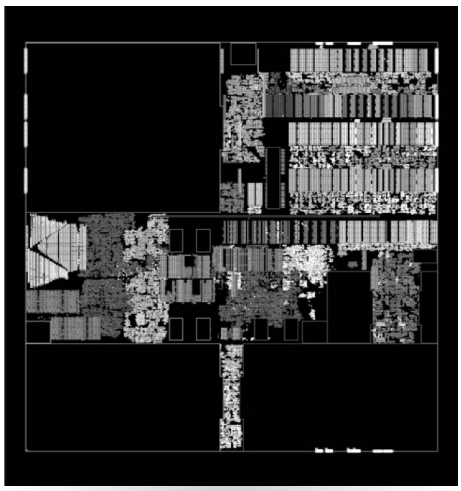

Figure 6. Raw tile—placed.

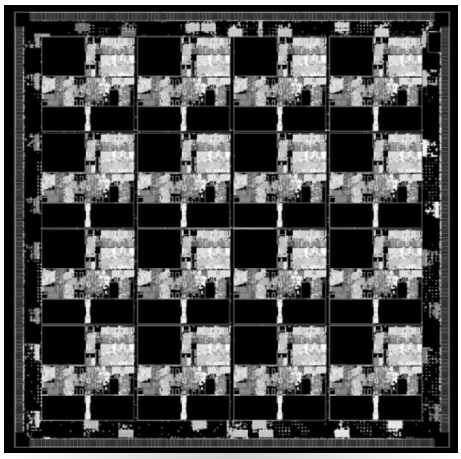

Figure 7. Raw chip—placed.

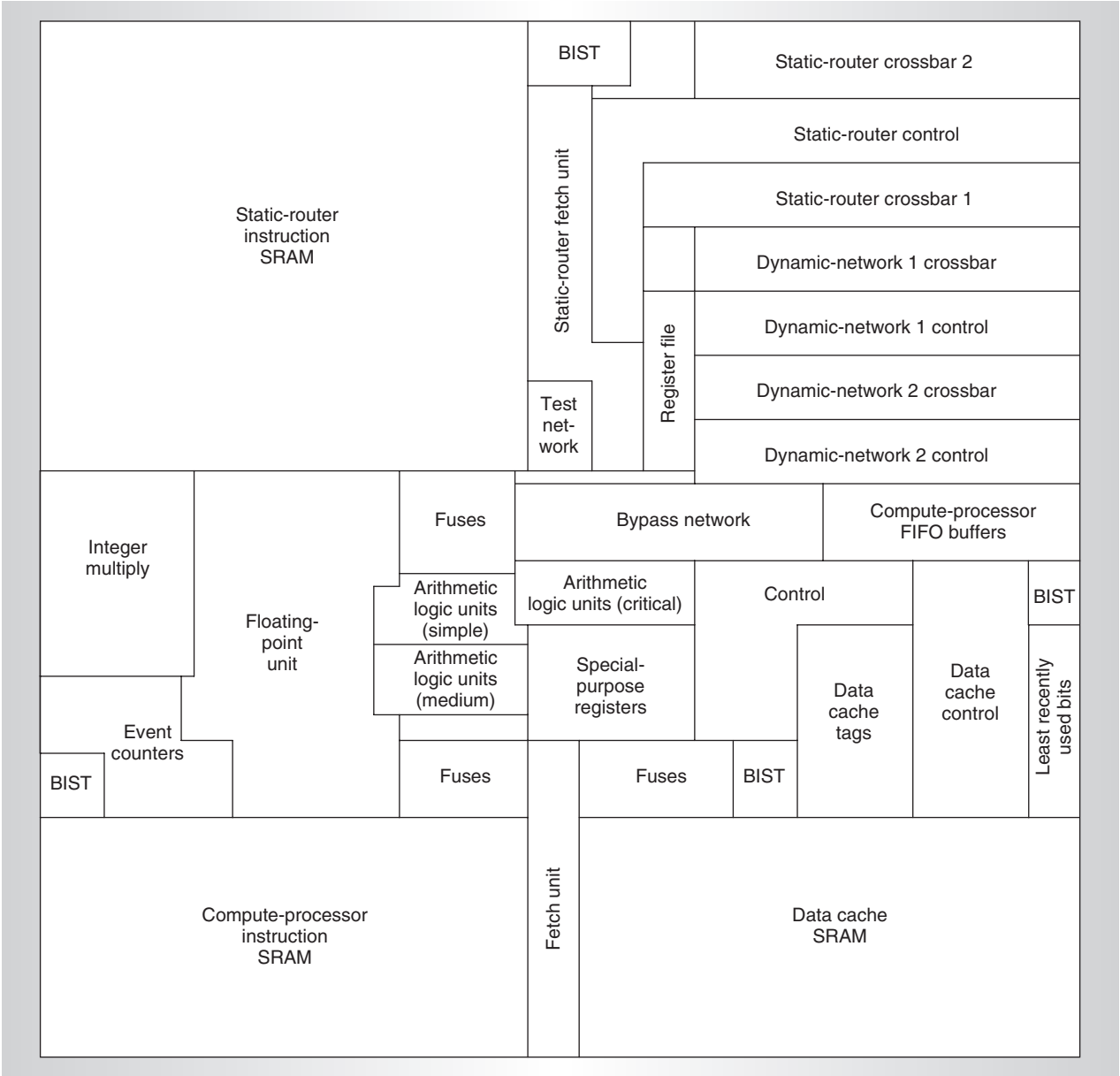

Figure 8. Raw tile floor plan.

the Raw motherboard to boot the gate-level Verilog and run test simulations.

Applications with a very small (two or three way) amount of instruction-level parallelism generally do not benefit much from running on Raw. This is because intertile latency is great enough that it is cheaper to compute locally than to distribute the computation to a neighbor tile. A two-way-issue compute processor would have helped fill out our parallelism profile for these applications, especially Specint benchmarks.

For scalar codes with a moderate degree of instruction-level parallelism, we found that our C and Fortran compiler, RawCC,[5] is effective at exploiting parallelism by automatically partitioning the program graph, placing the operations, and programming the routes for the static router. We attain speedups ranging from 6× to 11× versus a single tile on Specfp applications for a 16-tile Raw processor and 9× to 19× for 32 tiles. When parallelism is limited by the application, we find that RawCC comes close to the

hand-parallelized speedup, but tends to use up to two times as many tiles.

For structured applications with a lot of pipelined parallelism or heavy data movement like that found in software circuits, careful orchestration and layout of operations and network routes provided us with maximal performance because it maximizes per-tile performance. For these applications (and for the operating system code), we developed a version of the Gnu C compiler that lets the programmer specify the code and communication on a per-tile basis. Although this seems laborious, the alternative for these sorts of performance-oriented applications is an ASIC implementation, which is considerably more work than programming Raw. We are currently working on a new compiler that automates this mode of programming.[4]

The replicated tile design saved us considerable time in all phases of the project: design, RTL Verilog coding, resynthesis, verification, placement, and back-end flow.

Our design supports the glueless connection of up to 64 Raw chips in any rectangular mesh pattern, creating virtual Raw systems with up to 1,024 tiles. We intend to use this ability to investigate Raw processors with hundreds of tiles. We think that reaching the point at which a Raw tile is a relatively small portion of total computation could change the way we compute. We can imagine computation becoming inexpensive enough to dedicate entire tiles to prefetching, gathering profile data from neighbor tiles, translating (say for x86 emulation) and dynamically optimizing instructions, or even to simulating traditional hardware structures like video Ramdacs.

The idea of creating architectural analogs to pins, gates, and wires will ultimately lead to a class of chips that can address a great range of applications. It takes a lot of imagination to envision a 128-tile Raw processor, how fast a full-custom version would clock, or how a more sophisticated compute processor design could affect the overall system. It is our hope that the Raw research will provide insight for architects who are looking for new ways to build processors that leverage the vast resources and mitigate the considerable wire delays that loom on the horizon. MICRO

Acknowledgment

Raw is funded by DARPA, the National Science Foundation, and MIT's Project Oxygen.

References

1. R. Ho, K. Mai, and M. Horowitz, "The Future of Wires," *Proc. IEEE,* IEEE CS Press, Los Alamitos, Calif., April 2001, pp. 490-504.
2. Waingold et al., "Baring It All to Software: Raw Machines," *Computer*, vol. 30, no. 9, Sept. 1997, pp. 86-93.
3. T. Gross and D.R. O'Halloron, *iWarp, Anatomy of a Parallel Computing System*, MIT Press, Cambridge, Mass., 1998.
4. B. Thies et al., "StreamIT: A Compiler for Streaming Applications," tech. memo MIT-LCS-TM-620, Massachusetts Inst. Technology Lab. Comp. Sci., Cambridge, Mass., 2001; http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TM-620.pdf (current Feb. 2002).
5. Lee et al., "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine," *8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, ACM Press, New York, 1998, pp. 46-57.
6. J. Kubiatowicz, *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*, doctoral dissertation, EECS Dept., Massachusetts Inst. Technology, Cambridge, Mass., 1998.
7. M. Annaratone et al., "The Warp Computer: Architecture, Implementation and Performance," *IEEE Trans. Computers*, vol. 36, no. 12, 1987, pp. 1523-1538.
8. D. Shoemaker et al., "NuMesh: An Architecture Optimized for Scheduled Communication," *J. Supercomputing, vol. 10, no. 3, 1996*, pp. 285-302.
9. W. Dally, *A VLSI Architecture for Concurrent Data Structures*, Kluwer Academic Publishers, Boston, 1987.

Direct questions and comments about this article to Michael Taylor, MIT Laboratory for Computer Science, 200 Technology Square, Bldg. NE43, Cambridge, MA 02139; mtaylor@cag.lcs.mit.edu