The Raw Microprocessor:
A Computational Fabric for Software Circuits and General Purpose Programs

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

Laboratory for Computer Science
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 GB/s of I/O bandwidth, and has 2 MB of on-chip, distributed L1 SRAM memory, providing on-chip memory bandwidth of 43 GB/s. Is this the latest billion-dollar 3,000 man-year processor effort? In fact, Raw was designed and implemented by a handful of graduate students who had little or no experience in microprocessor implementation.

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

In this article, we demonstrate how the Raw research prototype uses a scalable ISA to attack the emerging wire delay problem by providing a parallel, software interface to the gate, wire, and pin resources of the chip.

We argue that an architecture that has direct, first class analogues to all of these physical resources will ultimately enable the programmer to extract the maximum amount of performance and energy efficiency in the face of wire delay. We show how existing architectural abstractions like 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.

Finally, we conclude with implementation details, a floorplan diagram and pictures of the design.

1.0 Technology Trends

Until recently, the abstraction of a wire as an instantaneous connection between transistors has shaped our assumptions and architectural designs. In an interesting twist, just as the clock frequency of processors has risen exponentially, the fraction of the chip that is reachable by a signal in a single clock cycle has at the same time been *decreasing* 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. [12]

Today, it takes on the order of 2 clock cycles to travel from edge-to-edge (about 15 mm) of a 2 gigahertz processor die. Processor manufacturers have been innovating to maintain high clock

rates in the face of the increased impact of wires. This innovation has been at many levels. The transition from aluminum to copper wires has reduced the resistivity and thus the RC 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 contain the effects of wire delay. Forced to worry about wire delays, designers place the logic elements of their processors very carefully, placing communicating transistors on critical paths very close to each other. Where once silicon area was precious, and logic was reused, logic is now freely duplicated (for example, the adders in the load/store unit and the ALU) to reduce wire lengths.

Even micro-architects are making concessions to wire delay. The architects of the 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. Later, the architects of the Pentium 4 were forced 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 are going to be experiencing latencies of 10 cycles or more across a processor die. It is going to become increasingly more challenging for existing architectures to turn chip resources into performance in the face of this wire delay.

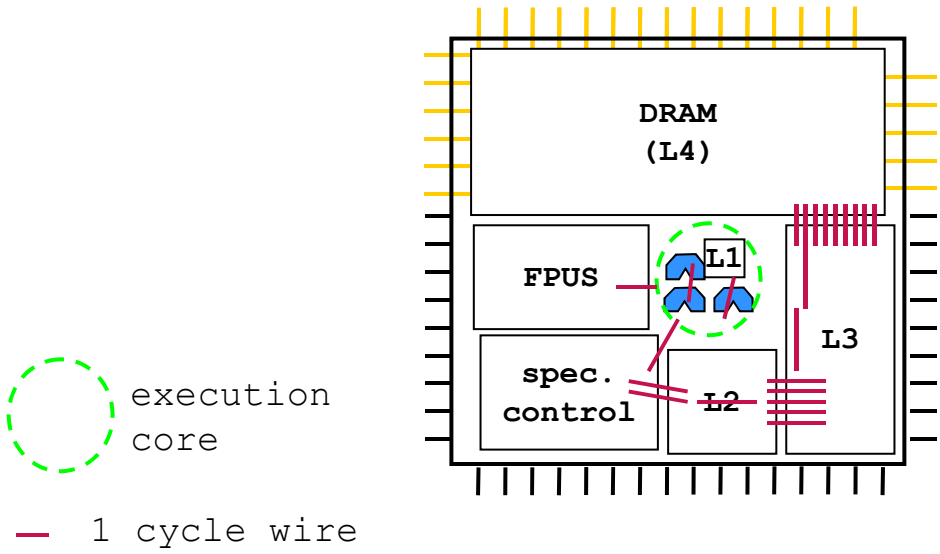

Figure 1: How today's micro-architectures will attempt to adapt as effective silicon area and pin resources increase, and as wire delay becomes worse

2.0 An Evolutionary Response for Current ISA's

Figure 1 shows our guess at how today's micro-architectures will attempt to adapt as effective silicon area and pin resources increase, and as wire delay becomes worse. Designers will want to utilize the increased silicon resources, while at the same time maintaining high clock rates. One

can imagine a high frequency "execution core", containing a number of nearby, clustered ALUs, with speculative control guessing which ALU cluster to issue to. Around this core is a host of pipelined "stuff" -- FPUs (which are less latency sensitive), pre-fetch engines, multilevel cache hierarchies, speculative control, and other out-of-band logic that's focused on making the tiny core in the middle run as efficiently and as fast as possible. And because most conventional ISAs do not have an architectural analogue to pins, the pins will mostly 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 designs. Even today, it is easy to tell that the percentage of silicon that is doing actual computation has been dropping quadratically over time. The 21464 design, an 8-way issue superscalar, is in excess of 27 times as large as the original 2-way issue 21064.[1] The "management" dwarfs the area occupied by ALUs, and the performance of these systems is getting increasingly non-deterministic and sensitive to the particulars of the program implementation. Intel, for instance, has produced a 300+ page document that suggests methods of avoiding stall conditions in the Pentium 4 micro-architecture. Furthermore, the power consumption, design and verification cost of these increasingly complex architectures is sky-rocketing.

3.0 The Raw Microprocessor

At MIT, we designed Raw to use a scalable ISA that provides a parallel software interface to the gate, wire, and pin resources of a chip. We think that an architecture that has direct, first class analogues to all of these physical resources will ultimately enable the programmer to 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 divides the usable silicon area into an array of 16 identical, programmable tiles. A tile contains an 8-stage in-order single-issue MIPS-derived processor, a 4-stage pipelined FPU, a 32 KB data cache, two types of communication switches -- static and dynamic, and 96 KB of memory that can be used for instructions and/or uncached local data. Each tile is sized so that the amount of time for a signal to travel through a small amount of logic and across the tile is one clock cycle. We expect that the Raw processors of the future will have hundreds or even thousands of tiles.

These tiles are interconnected by four 32-bit full-duplex on-chip networks, consisting of over 12,500 wires [see Figure 2]. Two of the networks are static (routes are specified at compile time) and two are dynamic (routes are specified at run time). Each tile is connected only to its four neighbors. Every wire is registered at the input to its destination tile. This means that *the longest*

1. Joel Emer's talk on the 21464 shows it being about 4.5 times as big as the 21264, and the 21264 is over 6 times as big as the 21064. Emer also states that the SMT capabilities of this machine do not use appreciably affect the die area.

wire in the system is no greater than the length or width of a tile. This property is very important for ensuring high clock speeds, and the continued scalability of the architecture.

These on-chip networks are exposed to the software through the Raw ISA, thereby giving the programmer the ability 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 chip. Effectively, the wire delay is exposed to the user as network hops. To go from corner to corner of the processor takes 6 hops, which corresponds to approximately six cycles of wire delay.

On the edges of the network, the network buses are multiplexed down onto the pins of the chip [see figure 3]. In order to toggle a pin, the user merely programs the on-chip network to route a value off the side of the array. Our 1657 pin CCGA (ceramic column-grid array) package provides us with fourteen full-duplex 7.2 Gb/s I/O ports at 225 MHz. This enormous pin budget often raises eyebrows among industrial microprocessor designers. The design does not require this many pins; rather the idea is to illustrate that, no matter how many pins a package has (100 or 100,000), Raw has a scalable architectural mechanism that will allow the programmer to put them to good use. Fewer pins merely require more multiplexing.

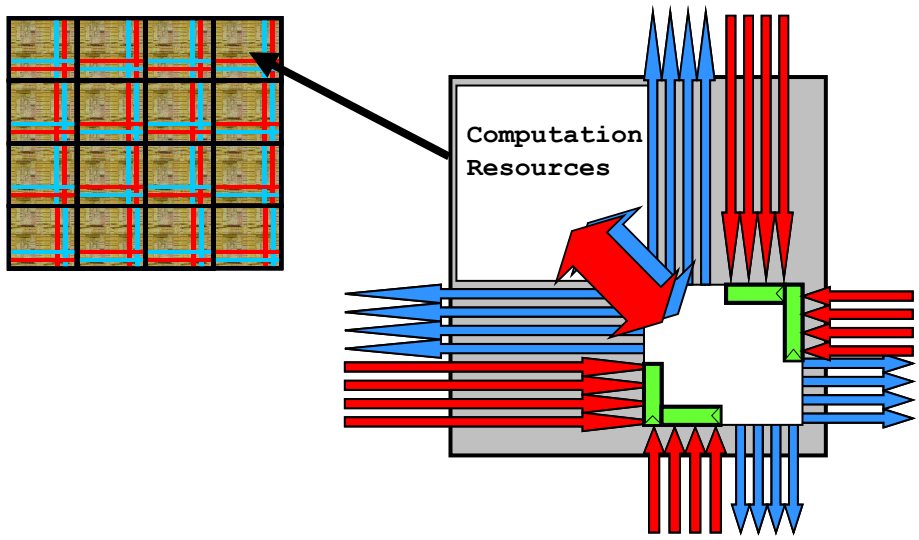

Figure 2: On-chip interconnect in Raw. The Raw microprocessor comprises 16 tiles. Each tile has some computational resources and four networks.

The Raw I/O port is a high-speed, simple (32 data pins, 2 control pins) and flexible word-oriented abstraction that allows the system designer to ratio the quantities of I/O devices according to the needs of the application domain. Memory intensive domains can have up to 14 dedicated interfaces to DRAM; other applications may not even 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). These devices can route through the on-chip networks to other devices in order to perform DMA accesses. Our intention is to hook up arrays of high speed data input devices, including wide-

word A/Ds, to experiment with Raw in domains that are extremely I/O, communication and compute intensive. In fact, one of our hardest problems is finding devices with interfaces that can produce data at the rates that we want.

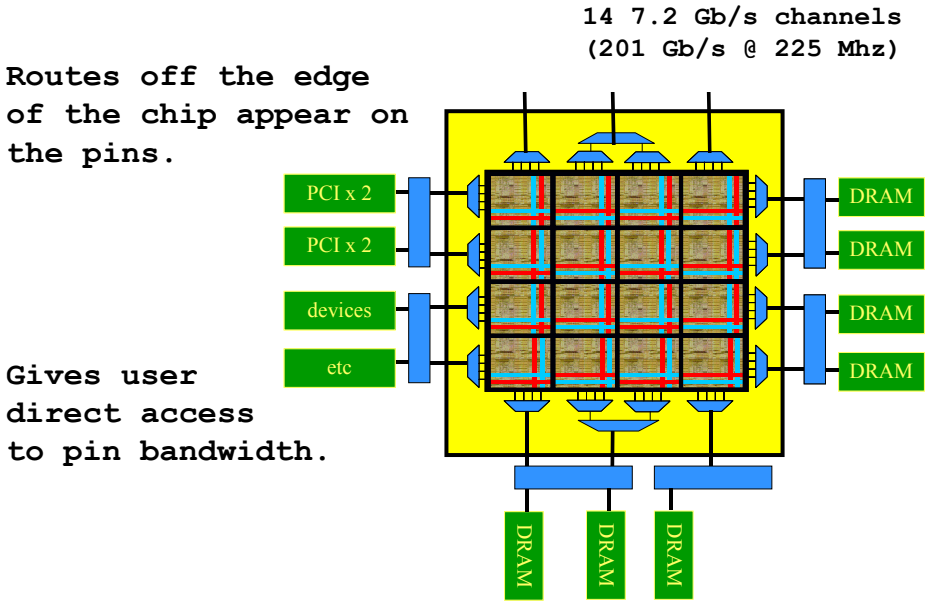

Figure 3: Pin multiplexing in Raw

Table 1: How Raw converts physical resources into architectural entities

Physical Entity	Raw ISA analogue	Conventional ISA Analogue
Gates	Tiles	Dynamic mapping of sequential program to small # of ALUs
Wire delay	Network Hops	Dynamic stalls for non-fastpath and mispredicted code
Pins	I/O ports	Speculative cache-miss handling (prefetching and large line sizes)

We believe that creating first class architectural analogues to the physical chip resources is a key technique for minimizing the ISA gap. The conventional superscalar 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!), we believe that the performance,. prominence of gate resources, wire delay, and pins merit the addition of corresponding architectural entities.

Table 1 contrasts the way the Raw ISA and conventional ISAs expose gates, wire delay, and pins to the programmer. Because the Raw ISA has more direct interfaces, we think Raw processors will have more functional units, and will have more flexible and more efficient pin utilization. High-end Raw processors will probably also 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 frequencies because of the explicit exposure of wire delay.

This exposure makes Raw very scalable. Creating subsequent, more powerful, generations of the processor is very 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, no global buses, and no structures that get larger as the tile or pin count increases. Finally, the longest wire, the design complexity, and the verification complexity are all independent of transistor count.

4.0 Application Domains for the Raw Microprocessor

The Raw microprocessor is designed to run computations that form a superset of those run on today's general purpose processors. Our goal is to run not just SpecInt and SpecFP, but word-level computations that require so much performance that they have been consigned to custom implementations on special purpose ASICs. We reason that if an application can take advantage of the customized placement and routing, the ample gates, and the programmable pin resources that is available in an ASIC process, it is likely that the application will benefit to some extent from the architectural versions of those same resources in the Raw microprocessor. First instance, our first-cut implementation of a software Gigabit IP router on a 225 MHz 16-tile Raw processor runs over 5 times faster than a hand-tuned implementation on a 700 MHz Pentium III processor. [11] Additionally, an implementation of video median filter on 128 tiles attained a 57 times 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, and the compilation process takes minutes, not months. Reflecting the Raw microprocessor's ASIC-like place and route facility, applications for Raw are often termed "software circuits." Sample software circuits with which we are experimenting include gigabit routers, video and audio processing, I/O protocols (RAID, SCSI, Firewire) and communications protocols (cell phones, multiple channel cellular base stations, HDTV, wireless bluetooth, 802.11b and 802.11a). These protocols could be running as a dedicated embedded host, or as a process on a general purpose machine.

At any point in time, a Raw processor can be running multiple processes simultaneously. A given process has been allocated some rectangular-shaped number of tiles (corresponding to "physical" threads which may themselves be virtualized) which correspond 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. This is because the physical threads of the process are likely to communicate and should be gang scheduled. Continuous or real-time applications can be "locked down" and will not be context switched.

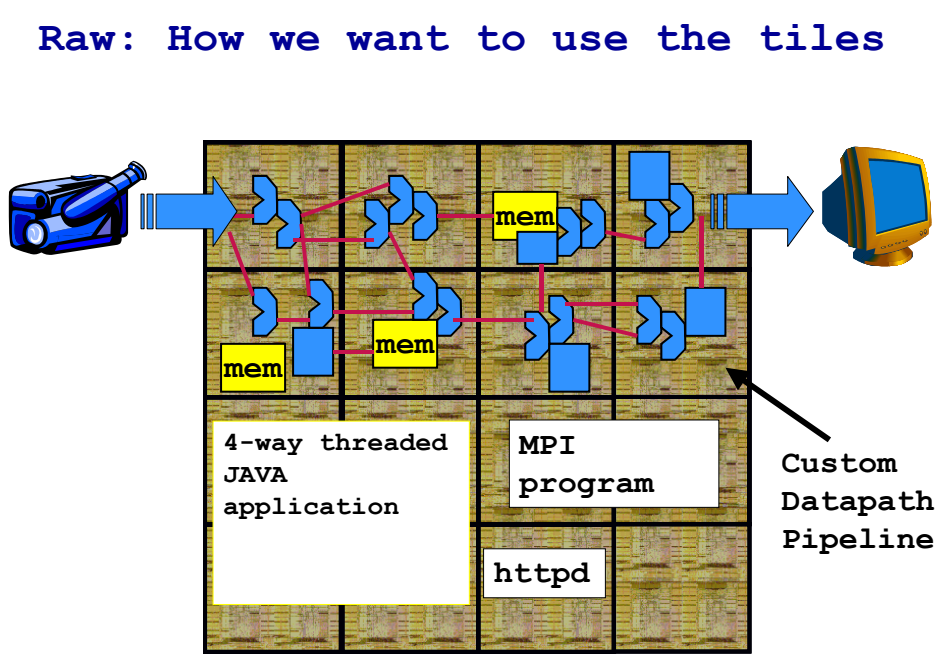

Figure 4: Application mapping onto a Raw microprocessor

Figure 4 shows a set of processes that are being space and time multiplexed on a Raw processor. Traditional applications like threaded java applications, MPI programs and server applications utilize Raw as a high-density multiprocessor. The top eight tiles in Figure 4 illustrate a more novel usage of the tiles. We are streaming in some video data over the pins, performing some sort of filter operation (the "software circuit") and streaming it out to a screen. In this situation, the tiles work together, parallelizing the computation. We assign operations to tiles in a fashion to minimize congestion and configure the network routes between these operations. This is very much like the process of designing a customized hardware circuit. In Raw, the customization is performed by the compiler [4, 5]. The customization can also be done by hand using Raw's assembly code.

In order to make all of this work, we need to have 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 a 1000 cycles, state of the art networks have latencies of 30 cycles [10]. The Raw network can route the output of the ALU on one tile to the input of the ALU of another tile in just 3 cycles. The networks are discussed in more detail shortly.

5.0 The Raw Tile

The Raw tile design crystallized around the idea of providing low-latency communication for efficient execution of software circuits. At the same time, we wanted to provide scalable versions of the standard toolbox of useful architectural constructs like data and instruction virtualization,

caching, interrupts, context switches, address spaces, latency-tolerance, and event counting. To achieve these goals, a Raw tile employs an 8-stage MIPS-derived single issue in-order pipeline (the "main processor"), a static routing processor that controls the two static networks, and a pair of dynamic routers. The static router manages the two static networks, which provides the low-latency communication required for software circuits and other applications with compile-time predictable communication. The dynamic routers manage the dynamic networks, which are used for unpredictable operations like interrupts, cache misses and unpredictable communication between tiles.

5.1 The Main Processor

Our goal in designing the main processor was to explore tightly-coupled network interfaces for processor pipelines. We wanted to make the network "first class" in every sense. This would maximize its utility. The most common network interfaces are memory mapped; other networks use special instructions for sending and receiving [7,10]. The most aggressive processor networks are register-mapped -- instructions can target the networks just as easily as registers [8].

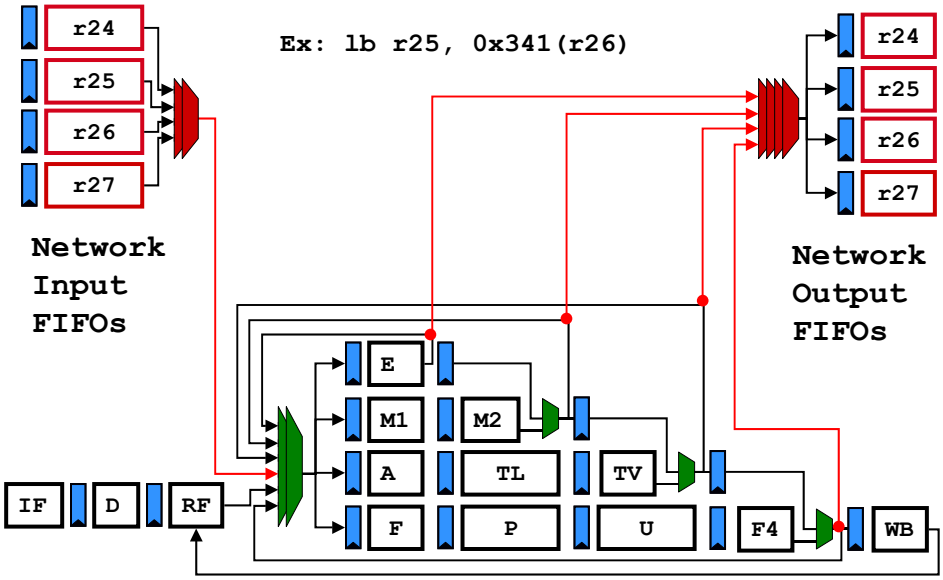

Figure 5: Tile processor pipeline

Our design takes network integration one step further: we integrate the networks directly into the bypass paths of the processor pipeline. This makes the network ports truly first-class citizens in the architecture. Figure 5 shows how this works. Registers 24..27 are mapped to the four physical networks on the chip. For example, a read from register 24 will actually pull an element from an input FIFO, while a write to register 24 will send the data word out onto that network. If data is not available on an input FIFO, or if an output FIFO does not have enough room to hold a result,

the processor will stall in the RF 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 $24. This gives the tile the option of keeping local copies of transmitted values.

The interesting activity occurs on the output FIFOs. Each output FIFOs is connected to each pipeline stage. The FIFOs pulls the oldest value out of the pipeline as soon as it is ready, rather than just at the writeback stage. This decreases the latency of an ALU-to-network instruction by as much as 4 cycles. 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.

This discussion of networks lends an interesting way of looking at a modern day processors. The register file used to be the central communication mechanism between functional units in a processor. Starting with the first pipelined processors, the bypass network has become largely responsible for the communication of active values, and the register file is more of a dumping ground or checkpointing facility for inactive values. The Raw networks are in some sense 2-D bypass networks serving as bridges between the bypass networks of separate tiles.

The early bypassing of values to the network has some challenging effects on the operation of the pipeline. Perhaps most importantly, it changes the semantics of the commit point of the processor. 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 latter stage. The simplest solution that we have found is to place the commit point at the execute stage. There is an interesting tension between responsiveness to the outside world, and the variety of performance optimization tricks that can be performed at the microarchitectural level.

5.2 The Static Routing Processor

For software circuits, we utilize the two static networks to route values between tiles. The goal of the static networks is to provide ordered, flow-controlled and reliable transfer of single word operands 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. They need to be the flow-controlled so that the program remains correct in the face of unpredictable architectural events like cache misses and interrupts.

The static routing processor is a 5-stage pipeline whose job is to control two routing crossbars and thus two physical networks. Each crossbar routes values between seven entities (the static router pipeline, North, East, South, West, Main Processor, and the other crossbar). The static router uses the same fetch unit design as the main processor, except it fetches a 64-bit instruction word from the 8 Kword instruction memory. This instruction simultaneously encodes a small command (conditional branches, accesses to a small register file, and decrements) and fourteen routes, one for each crossbar output. That's a total of fifteen operations per cycle per tile!

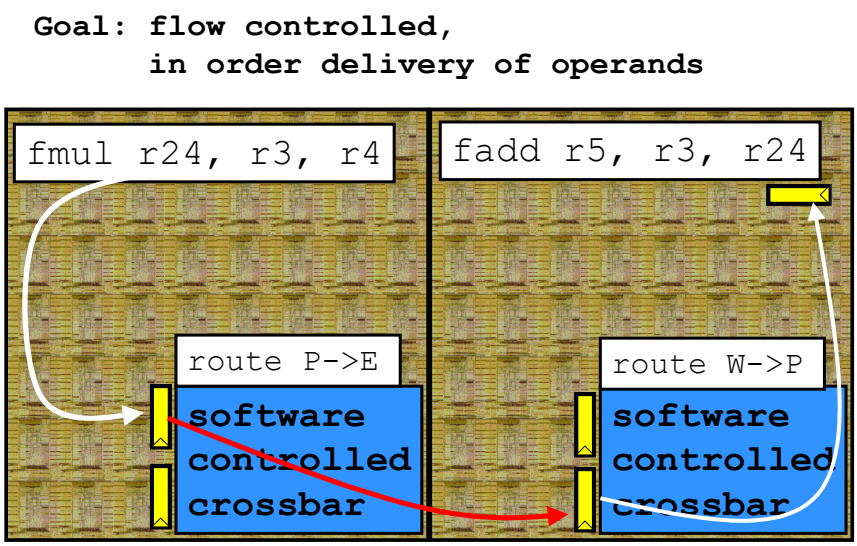

Figure 6: Two tiles communicating over the static network

For each word sent between tiles on the static network, there is a corresponding instruction in the instruction memory of each router that the word will travel through. These words are typically programmed at compile time, and are cached just like the instructions of the main processor. Because the static router knows what route will be performed long before the word arrives, the preparations for the route can be pipelined, and the data word can be routed immediately when it arrives. The static router provides single-cycle-per-hop latencies and can route two values in each direction per cycle. Figure 6 shows an example of a two tiles communicating across the static network.

The word leaving the FMUL instruction spends one cycle in each switch, and one cycle in the decode stage of the target tile, for a total latency of three cycles.

5.3 The Dynamic Networks

Early on in the Raw project, we realized the need for support for dynamic events as well as static events. This lead to the addition of a pair of dimension-ordered, wormhole-routed dynamic networks to the architecture [9]. To send a message on this network, the user injects a single header word that gives the destination tile (or I/O port), source tile, a user field and the length of the message. It 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 assumes that exploits the fact that most routes are straight.) On an uncongested network, the header will reach the destination in 2+X+1+Y+2 cycles -- two cycles of latency to leave the main processor (this counts as a hop and a turn), a number of X hops, one hop to turn (if X and Y != 0), a number of Y hops, and then a hop and a turn to enter the main processor.

One of the major concerns with dynamic network is deadlocks caused by the over-commitment of buffering resources. Classically, there are two solutions for deadlock: deadlock avoidance, which involves limiting usage to a set of disciplines that are known not to deadlock, and deadlock recovery, which involves draining the network to some source of copious memory when it appears to be deadlocked. Avoidance, unfortunately, restricts the generality of the usage of the network, while Recovery suffers from that fact that it depends on the existence of an out-of-band memory system (which would not exist if the memory system uses a network that employs deadlock recovery!). We found a relatively elegant solution -- we use a pair of networks, one (the "memory" network) has a very restricted usage model that uses deadlock avoidance, and the other (the "general" network) is unrestricted and uses deadlock recovery. If the general network deadlocks, it uses the memory network to recover.

The memory network is used by trusted clients -- operating system, data cache, interrupts, DMA, and I/O. These clients are each allocated a base number of unrestricted outstanding requests that they can have. For large, high performance transfers, the clients can negotiate permission with the operating system for larger quantities of outstanding requests to particular I/O ports.

The general network is used by untrusted clients who rely on the deadlock recovery mechanisms hardware to maintain forward progress when deadlock occurs. The software-implemented deadlock recovery code programs a configurable counter on the main processor to detect if words have been waiting for too long on the input [10]. 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. The general network is virtualized for each group of tiles that corresponds to a process. The upper left tile is considered to be "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.)

The memory network is shared by both software and the hardware caches. The hardware cache, on a cache miss, merely consults a configurable hash function to map addresses to destination ports. (The hash function approach is very elegant; it can even allow other tiles to service the cache misses.) It then issues header words like every other client of the network, and then a sequence of words that is interpreted by the DRAM controller. On a miss, it will also wait for the result and then transfers words into the cache memory. The DRAMs are just another I/O device on the network, and are equally accessible by hardware and software. We've found it useful on several occasions to write codes that bypass the cache and directly stream data into 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 to that tile. The tile's network hardware checks for that message and transparently pulls it off and sets the external interrupt bit. When the main processor services the interrupt, it will query the interrupt controller for the cause of the interrupt and then contact the appropriate device or DRAM.

6.0 Implementation

The Raw chip is a 16-tile prototype that we are developing using IBM's SA-27E .15 micron 6-layer ASIC copper process. Although a tile is only 4 mm x 4mm, we used an 18.2 mm x 18.2 mm die to allow us to use the high pin-count package. The 1657 CCGA (ceramic column grid array) package provides us with 1080 HSTL I/O pins. We estimate that the chip consumes 25 watts, mostly in memory accesses and pins. We quiesce unused functional units and memories and tri-state unused data I/O pins. We targeted a 225 MHz worst-case frequency (average case is typically %25 higher), which is competitive with other .15 micron ASIC processors, like Berkeley's IRAM, and Tensilica's customizeable processors.

Up to 64 chips to be combined into any rectangular mesh pattern to create virtual Raw systems of up to 1024 tiles (data words routed between chips will observe an extra three cycles of latency). We intend to use this ability to investigate Raw processors with hundreds of tiles. We think that reaching that point at which a Raw tile is a relatively small portion of total computation will totally change the way that we compute. One can imagine dedicating entire tiles to prefetching, to gathering profile data from neighbor tiles, to translating (say for x86 emulation) and dynamically optimizing instructions, or even to simulating traditional hardware structures like video RAM-DACs.

We pipelined our processor aggressively and treated control paths very conservatively in order to ensure that we would not have to spend significant periods closing timing in the backend. Despite this, we found that wire delay inside a tile was still large enough that placement could not be ignored. We created a library of routines (approximately 7000 lines of code) that automatically places the majority of the structured logic in the tile. This structured logic is clearly visible in the tile layout [see Figures 8]. This dropped the cycle time from 8 ns down to 4 ns, which matches Synopsys's timing estimates, and gave us a real feel for the size and structure of the architectural mechanism we had created. The screen captures of the chip and tile placement and the floorplan (which can be photocopied onto a transparency and overlayed) at the end of this paper show our handywork. The synthesis, backend processing, and placement infrastructure that we created can turn our RTL verilog source into a fully-placed chip in approximately 6 hours on one machine. We used a logic emulator donated by IKOS coupled with the Raw motherboard to boot the RTL verilog and run test simulations.

A difficult challenge for us was to keep ourselves from the temptations of making the absolutely highest performance, highest frequency tile processor, and instead concentrate on the research aspects of the project. The simplistic single-issue 8-stage pipeline uses static branch prediction and has a branch mispredict penalty of 3 cycles. A two-way issue main processor would have easily fit into the existing tile and met timing, and generally improved our performance numbers across the board. However, it would not have helped our research much.

We find that applications with a very small amount (two or three way) of instruction level parallelism (ILP) generally do not benefit much from running on Raw because the inter-tile latency is great enough that it is cheaper to compute locally then to distribute the computation to a neighbor tile. The two-way issue main processor, even just a simple VLIW, would have helped us fill out our parallelism profile for these applications, especially SpecInt benchmarks.

For fairly unstructured or legacy codes with a moderate degree of ILP, we found that our C and Fortran compiler, RawCC [4], is quite effective at exploiting parallelism by automatically partitioning the programming, placing the operations, and programming the routes on the static switch. We attain speedups of 6-11 versus a single tile on unmodified SpecFP applications for a 16-tile Raw processor, and 9-19 for a 32 tiles.When parallelism is limited by the application, we find that RawCC gets close to the hand-parallelized speedup, but tends to use up to two times as many tiles in doing so.

For structured applications with a lot of pipelined parallelism or heavy data movement like that found in software circuits, we've found that careful orchestration and layout of operations and network routes provides us with maximal performance because it maximizes the performance per tile. For these applications (and for the OS code), we've developed a version of gcc that allows the programmer to 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 to provide higher-level compilation tools for this mode of programming. [5].

The chip was a formidable learning experience for the Raw team overall. We found that the replicated tile design saved us considerable time in all phases of the project: design, RTL verilog coding, re-synthesis, verification, placement, and backend flow run-times.

7.0 Conclusion

Although it takes a certain imagination to envision a 128 tile Raw processor, or how fast a full-custom version will clock, or how a less simplistic main processor design will 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 are on the horizon. We feel that the idea of creating architectural analogues to pins, gates, and wires will ultimately lead to a class of chips that can truly address a greater range of applications. We think existing architectural abstractions like interrupts, caches, and virtualization can continue to be supported in this environment, even as new low-latency communication mechanisms like the static network enable effective orchestration of these gates.

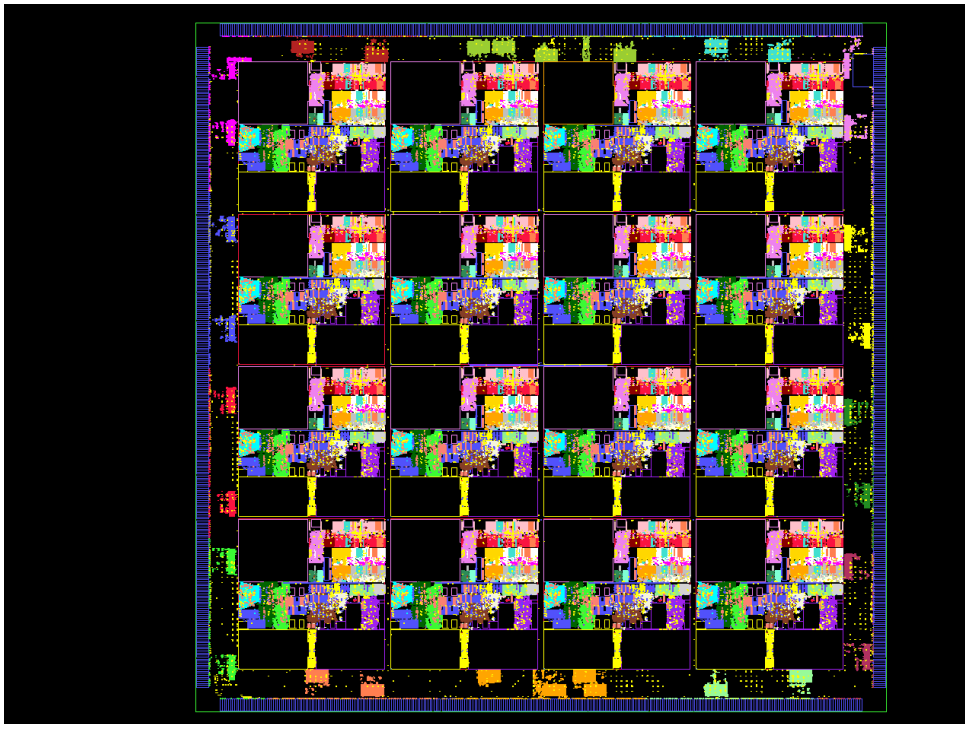

Figure 7: The Raw Chip - Placed

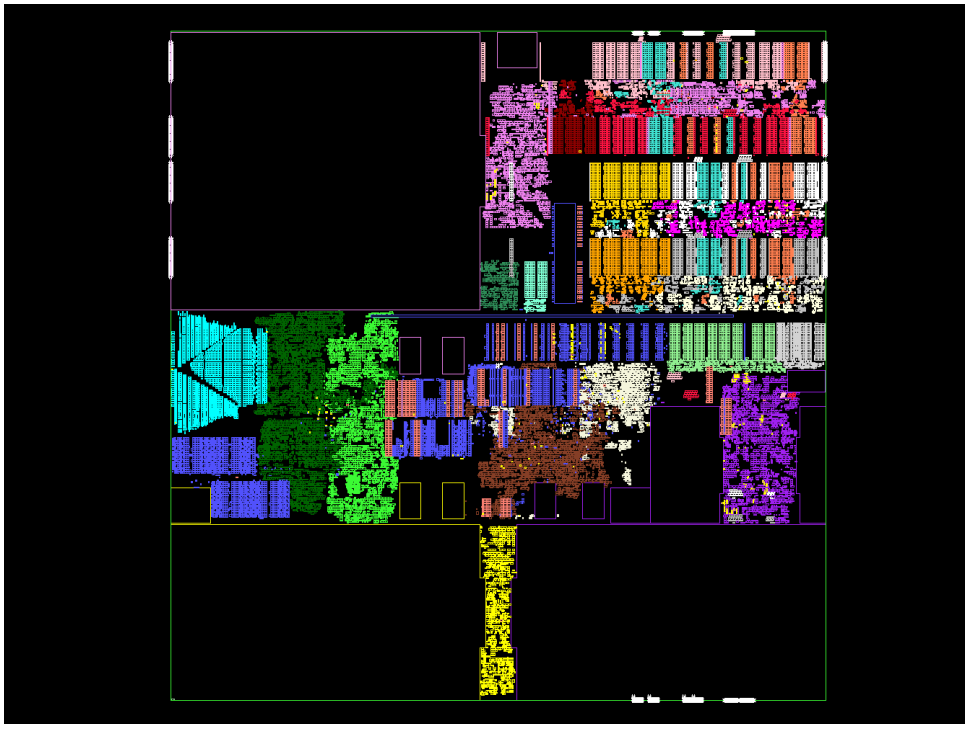

Figure 8: Raw Tile - Placed

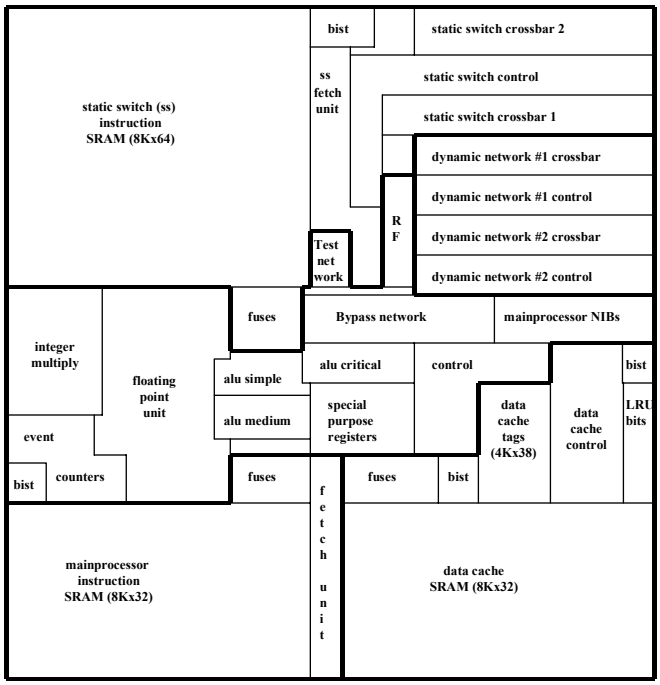

Figure 9: Tile Floorplan

[1] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. Proceedings of the IEEE, April 2001, pages 490-504.
[2] Waingold et al., Baring it all to Software: Raw Machines, IEEE Computer Magazine, September 1997
[4] Lee et al., Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine, ASPLOS-VIII, October 1998
[5] Thies et al., StreamIT: A Compiler for Streaming Applications, MIT/LCS TM LCS-TM-620, August 2001.
[7] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica Lam, Onat Menzilicioglu, and Jon A. Webb, "The Warp computer: Architecture, implementation and performance," IEEE Transactions on Computers, 36(12):1523--1538, December 1987.
[8] Thomas Gross and David R. O'Halloron. iWarp, Anatomy of a Parallel Computing System. The MIT Press. Cambridge, MA 1998.
[9] Dally, A VLSI Architecture for Concurrent Data Structures, Kluwer Academic Publishers, 1987
[10] Kubiatowicz, Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor, PhD thesis, MIT, 1998.
[11] Gleb Chuvpilo, David Wentzlaff, and Saman Amarasinghe. Gigabit IP Routing on Raw. In Proceedings of the 8th International Symposium on High-Performance Computer Architecture, Workshop on Network Processors, February 2002.
[12] R. Nagarajan, K. Sankaralingam, D.C. Burger, and S.W. Keckler. "A Design Space Evaluation of Grid Processor Architectures." 34th International Symposium on Microarchitecture (MICRO), pp. 40-51, December, 2001.