Theme Feature

Baring It All to Software: Raw Machines

This innovative approach eliminates the traditional instruction set interface and instead exposes the details of a simple replicated architecture directly to the compiler. This allows the compiler to customize the hardware to each application.

Elliot Waingold
Michael Taylor
Devabhaktuni Srikrishna
Vivek Sarkar
Walter Lee
Victor Lee
Jang Kim
Matthew Frank
Peter Finch
Rajeev Barua
Jonathan Babb
Saman Amarasinghe
Anant Agarwal
Massachusetts Institute of Technology, Laboratory for Computer Science

As our industry develops the technology that will permit a billion transistors on a chip, computer architects must face three converging forces: the need to keep internal chip wires short so that clock speed scales with feature size; the economic constraints of quickly verifying new designs; and changing application workloads that emphasize stream-based multimedia computations.

One approach is to rely on a simple, highly parallel VLSI architecture that fully exposes the hardware architecture's low-level details to the compiler. This allows the compiler—or, more generally, the software—to determine and implement the best resource allocation for each application. We call systems based on this approach Raw architectures because they implement only a minimal set of mechanisms in hardware. Raw machines require only short wires, are much simpler to design than today's superscalars, and support efficient pipelined parallelism for multimedia applications.

RAW ARCHITECTURE

Our general philosophy is to build an architecture based on replicating a simple tile, each with its own instruction stream. As Figure 1 shows, a Raw microprocessor is a set of interconnected tiles, each of which contains instruction and data memories, an arithmetic logic unit, registers, configurable logic, and a programmable switch that supports both dynamic and compiler-orchestrated static routing.

The tiles are connected with programmable, tightly integrated interconnects. The tightly integrated, synchronous network interface of a Raw architecture allows for intertile communication with short latencies similar to those of register accesses. Static scheduling guarantees that operands are available when needed, eliminating the need for explicit synchronization.

In addition, each tile supports multigranular (bit-, byte- and word-level) operations and programmers can use the configurable logic in each tile to construct operations uniquely suited to a particular application. Together, these features will enable high switching speeds and dramatically simplify hardware design and verification tasks.

Small replicated tiles

As Figure 1 shows, a Raw machine is made up of a set of interconnected tiles. Each tile contains a simple, RISC-like pipeline and is interconnected with other tiles over a pipelined, point-to-point network. Having many distributed registers eliminates the small-register name-space problem, allowing a greater degree of instruction-level parallelism (ILP).

Static RAM (SRAM) distributed across the tiles eliminates the memory bandwidth bottleneck and provides significantly shorter latency to each memory module. The distributed architecture also allows multiple high-bandwidth paths to external Rambus-like DRAM—as many as packaging technology will permit. A typical Raw system might include a Raw microprocessor coupled with off-chip memory and stream-IO devices. The amount of memory is chosen to roughly balance the areas devoted to processing and memory and to match the memory-access time to the processor clock. The compiler implements higher level abstractions like caching, global shared memory, and memory protection.[1] Software handles dynamic events like cache misses.

Unlike current superscalars, a Raw processor does not bind specialized logic structures such as register-renaming logic or dynamic instruction-issue logic into hardware. Instead, the focus is on keeping each tile small to maximize the number of tiles that can fit on a chip, increasing the chip's achievable clock speed and the amount of parallelism it can exploit. For example, a single one-billion-transistor die using a conventional logic technology could carry 128 tiles. Each tile would use 5 million transistors for memory, splitting them among a 16-Kbyte instruction memory (IMEM); a 16-Kbyte switch instruction memory (SMEM); and a 32-Kbyte

0018-9162/97/$10.00 © 1997 IEEE

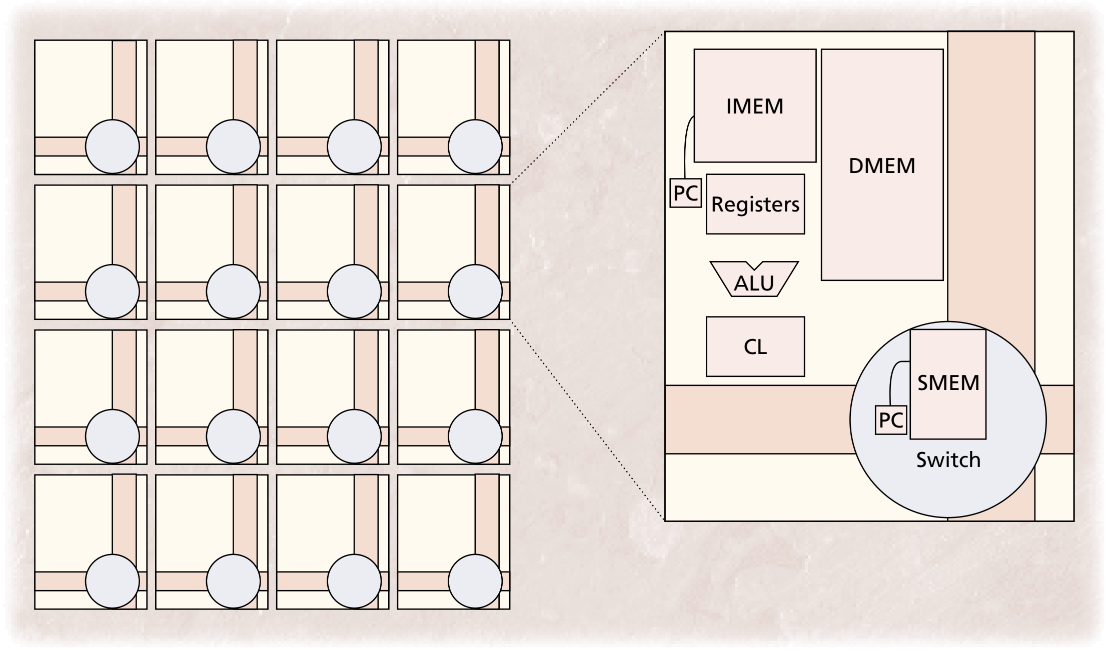

Figure 1. A Raw processor is constructed of multiple identical tiles. Each tile contains instruction memory (IMEM), data memories (DMEM), an arithmetic logic unit (ALU), registers, configurable logic (CL), and a programmable switch with its associated instruction memory (SMEM).

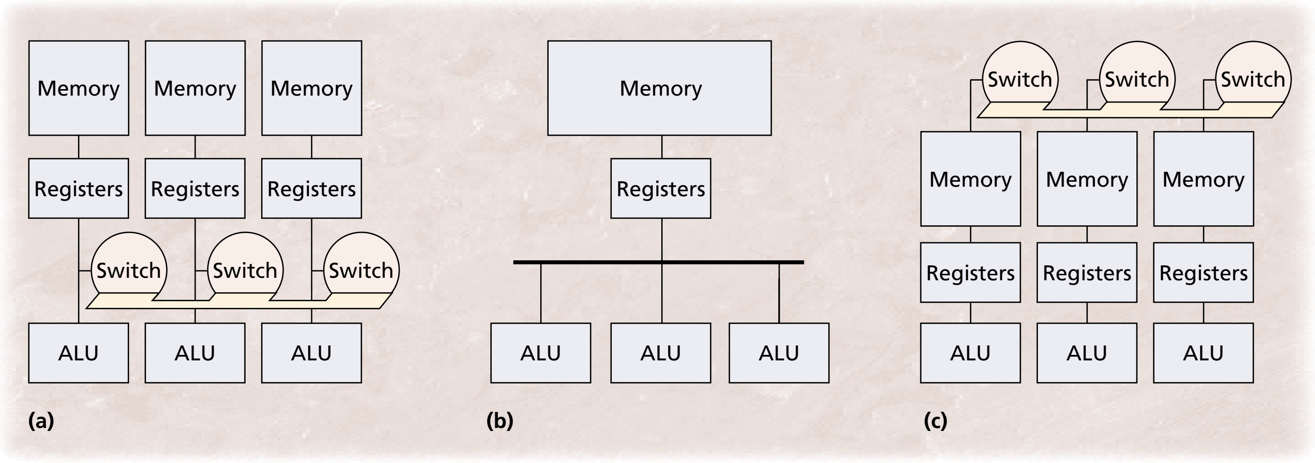

Figure 2. Raw microprocessors differ from superscalar and multiprocessor architectures. (a) Raw microprocessors distribute the register file and memory ports and communicate between ALUs on a switched, point-to-point interconnect. (b) A superscalar contains a single register file and memory port and communicates between ALUs on a global bus. (c) Multiprocessors communicate at a much coarser grain through the memory subsystem.

first-level data memory. Each type of memory is implemented with 6-transistor SRAM memory cells backed by 128 Kbytes of dynamic memory implemented using 2-transistor memory cells. Each tile could devote a generous 2-million-transistor equivalent area to the pipelined data path and control (say, an R2000-equivalent CPU, floating-point unit, and configurable logic). Interconnects would consume about 30 percent of the chip area.

As Figure 2 shows, a Raw architecture basically replaces a superscalar processor's bus architecture with a switched interconnect while the software system implements operations such as register renaming, instruction scheduling, and dependency checking. Reducing hardware support for these operations opposes current trends, but it makes more chip area available for memory and computational logic, results in a faster clock, and reduces verification complexity. Taken together, these benefits can make the software synthesis of complex operations competitive with hardware for overall application performance.

Programmable, Integrated Interconnect

As Figure 2 shows, a Raw machine uses a switched interconnect instead of buses. The switch is integrated directly into the processor pipeline to support single-cycle message injection and receive operations. The processor communicates with the switch using dis-

Compilers can schedule single-word data transfers and exploit ILP, in addition to coarser forms of parallelism.

tinct opcodes to distinguish between accesses to static and dynamic network ports. No signal in a Raw processor travels more than a single tile width within a clock cycle.

Because the interconnect allows intertile communication to occur at nearly the same speed as a register read, compilers can schedule single-word data transfers and exploit ILP, in addition to coarser forms of parallelism. Integrated interconnects also allow channels with hundreds of wires instead of tens—very large scale integration switches are pad-limited and rarely dominated by internal switch area. The switch multiplexes two logically distinct networks—one static and one dynamic—over the same set of physical wires. The dynamic wormhole router makes routing decisions based on each message's header, which includes additional lines for flow control.

Control

Each tile includes two sets of control logic and instruction memories. The first set controls the operation of the processing elements. The second is dedicated to sequencing routing instructions for the static switch. Separate controls for processor and switch lets the processor take arbitrary, data-dependent branches without disturbing the routing of independent messages passing through the switch. Loading a different program into the switch instruction memory changes the switch schedule. Programming network switches with compilation time schedules lets the compiler statically schedule the computations in each tile. Static scheduling eliminates synchronization and its significant overhead.

Compiler orchestration ensures that the static switch stalls infrequently. Dynamic events, however, may force one of the tiles to delay its static schedule. For correctness, the static network provides minimal hardware-supported flow control. Although both static and dynamic networks export flow-control bits to the software, we are exploring strategies to entirely avoid the flow-control requirement on the static network.

Comparing Raw to Other Architectures

The Raw approach builds on several previous architectures. A Raw architecture seeks to execute pipelined applications (like signal processing) efficiently, as did earlier systolic-array architectures. Like computers based on field-programmable gate arrays (FPGAs), a Raw machine permits the construction of application-specific custom operations and communication schedules. Finally, like very long instruction word (VLIW) processors, a Raw processor simplifies and exposes the instruction-scheduling hardware to the compiler.

Systolic arrays

The design of systolic-array architectures like iWarp[1] and NuMesh[2] emphasizes efficient processing of streams of data. They share with the Raw approach the philosophy of building point-to-point networks that support static scheduling. However, the cost of initiating a message in iWarp or introducing a new pipeline pattern in NuMesh is too high. This limited compilers to focusing on signal processing applications that have a uniform structure, in order to amortize startup costs over long messages. Static scheduling in coarse-grained environments like those provided by iWarp and NuMesh is very difficult—predicting static events over long periods of time is hard. So, a Raw architecture, which also handles fine-grained parallelism, provides several advantages. The register-like communication latency in a Raw processor allows it to exploit the same types of ILP that superscalar processors exploit. Predicting latencies over short instruction sequences and statically scheduling multiple, tightly coupled instruction streams is much simpler.

FPGAs

A second approach to processing streams is to build computers from FPGAs. Like custom hardware, these chips can be configured to user specification. FPGA-based machines achieve their speeds by exploiting fine-grained parallelism and fast static communication. An FPGA's software has access to its low-level details, allowing the software to optimize mapping of the user application. Users can also bind commonly used instruction sequences into configurable logic. As a result, these special-purpose instruction sequences can execute in a single cycle. A Raw machine also incorporates these features.

FPGA systems, however, do not support instruction sequencing and are thus inflexible—they require loading an entire bitstream to reprogram the FPGAs for a new operation. Compilation for FPGAs is also slow because of their fine granularity. The onerous compilation times of our FPGA-based Raw prototype (discussed later) amply demonstrate this compilation speed problem.

Unlike FPGA systems, however, Raw machines support instruction sequencing. They are more flexible, merely pointing to a new instruction to execute a new operation. Compilation in Raw is fast because the hardware contains commonly used compute mechanisms such as ALUs and memory paths. This eliminates repeated, low-level compilations of these units. Binding common mechanisms into hardware also yields faster execution speed, lower area, and better power efficiency than FPGA systems.

VLIW processors

Work in very long instruction word processors inspires many features of the Raw approach. Like the VLIW Multiflow Trace machine,[3] Raw machines have a large register name space, a distributed register file, and multiple memory ports. Both

Multigranular operations

A Raw architecture is multigranular—it implements wide-word arithmetic logic units and multiple-bit or byte-level operations in each tile. Thus, a Raw processor is coarser than traditional FPGA-based processors but, for a given application, can achieve the same level of fine-grained parallelism. Logic simulation, for example, requires bit-level operations, so FPGA-based emulators are simply special-purpose processors that handle bit-level parallelism.

The byte-granular configurable logic permits the ALUs to be used for either a few wide-word operations or many narrow-word computations much like HP's MAX, Sun's VIS or Intel's MMX. The small amount of configurable logic in each Raw tile—for which software can select the data path width—permits a Raw processor to support multigranular operation.

Configurability

Raw architectures combine bit- or byte-level parallelism with special communication paths among the individual bits or bytes. This permits significantly more powerful multigranular operations than those supplied by MMX-like instruction sets. We can view configurable logic as a means for the compiler to create customized instructions without resorting to longer software sequences. In the Conway's Game of Life benchmark discussed later, configuration compresses a software sequence from 22 cycles into one.

SOFTWARE SUPPORT

The software system can leverage Raw's high degree of parallelism and wide, static network for algorithms that require high-bandwidth and fine-grained communication. Thus, a Raw architecture is particularly suited for traditional scientific applications and for processing streams of data.

In the past, it has been difficult to compile for a static architecture like Raw. Today's workloads, however, are beginning to emphasize stream-processing problems that can benefit significantly from the type of static pipelining Raw architectures support. In addition,

Stream-processing problems can benefit significantly from the type of static pipelining Raw architectures support.

machines rely heavily on compiler technology to discover and statically schedule ILP. Unlike traditional VLIW processors, however, Raw machines provide multiple instruction streams. Individual instruction streams allow Raw processors to perform independent but statically scheduled computations (such as loops with different bounds) in different tiles.

Multiscalar processors

The Raw processor is deceptively similar to a multiscalar processor, but the latter does not expose all its hardware resources to software. For example, a multiscalar might expose only 32 registers through a compact ISA, but relegate register renaming and dependence checking to hardware. A Raw machine exposes all registers to the compiler, and it implements similar mechanisms in software. Clearly, full exposure affords the compiler more flexibility in using the registers.

For example, a Raw machine allows explicit forwarding of register values to specific tiles, which permits the use of a scalable mesh interconnect to couple the tiles. In contrast, a multiscalar must use a bus (possibly pipelined) to broadcast forwarded values to each tile. Of course, the Raw approach's drawback is that it must revert to a software-simulated broadcast when the compiler cannot statically establish a schedule.

Single-chip multiprocessor

It is natural to compare Raw architecture with the logical evolution of multiprocessors: on-chip integration of a multiprocessor built from simple RISC processors.[4] Like a Raw machine, such a multiprocessor uses a simple replicated tile and provides distributed memory. But unlike a Raw processor, the cost of message startup and synchronization hampers the multiprocessor's ability to exploit fine-grained ILP.

IRAM architectures

Raw architectures also promise better on-chip balance than the processor-in-memory or Intelligent RAM architectures, which integrate processor and memory on the same chip to improve performance. As Dick Sites observed, in many applications, current superscalars spend three out of four CPU cycles waiting for memory.[5] Two reasons for this are long memory latencies and limited memory bandwidth, which can be somewhat relieved by on-chip integration. IRAM chips, however, will have longer delays than Raw chips because of long memory bit-lines or long crossbar wires in a multibanked memory. These delays will be less tolerable when compared to the faster processing and switching speeds of future-generation chips.

References

1. S. Borkar et al., "Supporting Systolic and Memory Communication in iWarp," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 70-81.
2. D. Shoemaker et al., "NuMesh: An Architecture Optimized for Scheduled Communication," *J. Supercomputing*, 1996, pp. 285-302.
3. J.A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proc. 10th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1983, pp. 140-150.
4. K. Olukotun et al., "The Case for a Single-Chip Multiprocessor," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems VII*, ACM Press, New York, 1996, pp. 2-11.
5. R. Sites, "Architects Look to the Future," *Microprocessor Report*, Aug. 5, 1996, pp. 19-20.

A key issue in handling dynamic events is to maximize the independence of the dynamic events from other static parts of the program.

runtime systems can provide extra dynamic support when the compiler cannot easily identify parallelism in, for example, programs that use pointer-based data structures. In this case, the compiler identifies threads to speculatively execute in parallel. It will also construct software checks for resolving dependencies between threads, thereby making full compilation-time knowledge of such dependencies unnecessary. Runtime checks are slower than corresponding hardware mechanisms, but the compiler can optimize individual checks whenever information becomes available.

Compilation

A compiler for Raw processors must take a single-threaded (sequential) or multithreaded (parallel) program written in a high-level programming language and map it onto Raw hardware. A Raw architecture's explicitly parallel model is different from that of a superscalar. A superscalar's hardware extracts instructions for concurrent execution from a sequential instruction stream. Unlike the coarse-grained parallelism exploited on multiprocessor systems, the Raw compiler views the set of *N* tiles in a Raw machine as a collection of functional units for exploiting ILP.

Partitioning, placement, routing, and scheduling. An early phase of the compiler identifies and partitions for fine-grained ILP[2] by balancing the benefits of parallelism versus the overheads of communication and synchronization. Because these overheads are much lower than in traditional multiprocessors, partitioning can occur at a much finer grain than in conventional parallelizing compilers. Partitioning generates parallel code for an idealized Raw machine with the same number of tiles as the physical architecture. This phase, however, assumes an idealized, fully connected switch; an unbounded number of virtual registers per tile; and symbolic data references.

Placement selects a one-to-one mapping from threads to physical tiles. The placement algorithm minimizes a latency and bandwidth cost measure and is a variant of a VLSI cell-placement algorithm.[3]

Finally, routing and global scheduling allocates physical network resources to minimize the overall program completion time. This phase produces a program for each tile and switch. We plan to adapt the Topology Independent Pipelined Routing and Scheduling algorithm[4] for this task. TIERS is a VLSI algorithm that uses a greedy allocation scheme. It determines a time-space path for each intertile communication event and a schedule for the instructions in each thread.

Configuration selection. An additional compilation phase selects an application-specific configuration for loading into the configurable logic. For each custom operation, the configuration phase must both output a specification for the configurable logic and rewrite the intermediate code. This replaces each compound operation by a call to the appropriate custom instruction. The compiler will invoke a logic synthesis tool to translate a custom operation specification into the appropriate bit sequence for the configurable logic. Commonly occurring subtree patterns can be good candidates for implementation as custom instructions. To identify these patterns, the compiler will extend the dynamic-programming algorithms used in tree pattern-matching systems.[5]

Dynamic-event support

Dynamic events typically occur when the compiler cannot resolve data dependencies or latencies at compilation. In this section we discuss a number of techniques that the compiler can use to change dynamic patterns into static patterns. When static techniques fail, dynamic software methods are the next resort. For programs that the compiler cannot analyze, we plan to implement a system similar to the Multiscalar,[7] but with software-based runtime checks. We believe many of these checks can be eliminated because the compiler has full access to the underlying hardware mechanisms. Hardware implementation is a last resort. A key issue in handling dynamic events is to maximize the independence of the dynamic events from other static parts of the program.

Software dynamic routing. A Raw system uses one of two approaches to handle dynamic messages (messages that cannot be routed and scheduled at compilation). The first is a software approach that statically reserves channel bandwidth between nodes that may potentially communicate. This approach preserves the program's predictability by using a static, data-independent routing protocol. In the worst case, this conservative approach will involve an all-to-all personal communication schedule between the processing elements.[6]

The second approach uses the dynamically routed network. By conservatively estimating the delivery time of dynamic messages and allowing enough slack, the compiler can still hope to meet its global static schedules. Software checks of the flow control bits preserve correctness in this approach.

Software memory dependency checking. Dynamically checking for dependencies between every pair of memory operations is expensive. Most superscalar processors can sustain only one or at most two memory instruction issues per cycle (as does the Digital Alpha 21264), limiting the total parallelism they can achieve. For applications that require high-bandwidth runtime dependency checking, the compiler can construct distributed software dependency-checking structures from multiple Raw tiles. As with directory based cache-coherence protocols, the system can statically assign the task of resolving the dependencies for each address to a different system node. Previous research gives an example of an all-software, cache-coherent, shared memory.[1]

Although such a software-implemented system is less efficient than an all-hardware implemention, the exposed mechanisms in a Raw architecture provide opportunities for optimization that are unavailable in all-hardware implementations. Consider the loop

```
for (i = 0; i < n; i++)
     a[b[i]] = a[c[i]]
```

Figure 3 shows a systolic structure that maintains all the true value dependencies of this loop but allows independent memory references to proceed in parallel. Because the compiler knows that accesses to the a, b, and c arrays are independent, it can create a 6-tile systolic structure. Each box represents a Raw tile. A value is read out of the b and c arrays every cycle, while the tile handling the a array alternates each cycle between handling a load or a store. The limit is a potential value dependence between each load from the a array and the store in the following cycle. This implementation performs an iteration in two cycles without requiring dual-ported caches. Providing hardware for dynamic dependency checking is less important because a Raw compiler can provide memory dependence information to the hardware.

Logical data partitioning. The compiler can further reduce the required dynamic software support by creatively partitioning data. The offset for memory operations is often known at compilation time (for example, the stack frame offset for a local variable or the field/element offset in a heap-allocated object) even though the base address is unknown. Therefore, by using low-order interleaving and $N \times B$ alignment we can calculate the destination tile for each memory request.

- *Low-order interleaving.* We assume that the address space is divided into blocks of $B = 2^b$ bytes (typical values of B are 8, 16, 32, and 64), and that the addressing of blocks is low-order-interleaved across tiles. The compiler chooses larger block sizes when it expects a high level of spatial locality in data accesses. This means that the bit layout of a data address contains the block address in the node/tile, n bits for the node/tile ID, and b bits for the block offset. The number of tiles is $N = 2^n$.
- *$N \times B$ alignment.* We assume the base address of commonly used program entities (stack frames, static arrays, heap-allocated structures, and so on) is aligned with an $N \times B$ boundary (that is, has zeroes in the lowest $n + b$ bits). This enables the compiler to statically predict the destination tile for a known offset, even though the actual base address is unknown. If the system implements some form of virtual memory, then this scheme will continue to work if $N \times B$ is less than the page size.

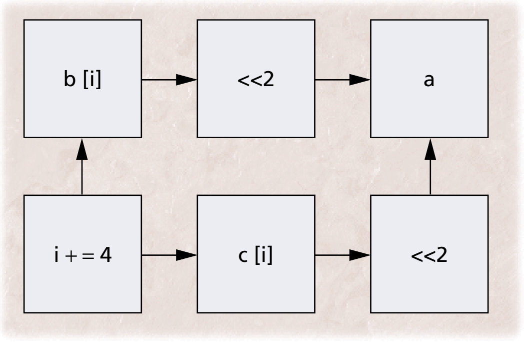

Figure 3. Six-tile systolic structure that maintains value dependencies for a looped array access.

For base pointers such as function parameters that may not be known at compilation time to be $N \times B$ aligned, we suggest generating dual-path (multiversion) code with a runtime test that checks whether unknown base addresses are $N \times B$ aligned.

RAWLOGIC PROTOTYPE RESULTS

We wanted to learn how to write compilers that can statically orchestrate the communication and computation in multiple threads as well as study the performance of the Raw processor. We thus implemented the RawLogic prototype and an associated compilation system by leveraging commercial FPGA-based logic emulation technology. The RawLogic prototype consists of a VirtuaLogic Emulator from Ikos Systems coupled with a Sun SparcStation 10/51 via an Sbus interface card from Dawn VME Products. The emulator also has a SCSI interface for downloading configurations and controlling clock speed. Our system has five boards, each with 64 directly connected Xilinx 4013 FPGAs.

RawLogic supports only some Raw architecture features. It has simple replicated tiles and supports statically scheduled, tightly integrated communication, multigranularity, and configurability. But it does not support the instruction processing of a more general Raw system. Rather, we converted each static control sequence into an individual state machine and hardwired it into RawLogic. RawLogic therefore has the problems associated with FPGA-based systems—it lacks flexibility and has long compilation times.

To compile applications for RawLogic, we developed a framework that specifies the dependence structure of a program's loops (in C) and the computation it performs (in behavioral Verilog). This program specification is used to automatically generate a behavioral Verilog netlist for the program. A commercial behavioral compiler automatically synthesizes a gate-level netlist, which is then processed by a VirtualWires compiler. This compiler partitions, places, and schedules the

Table 1. Benchmark results for the Raw processor.

Benchmark category	Benchmark name	Data width (bits)	No. of elements	No. of gates (thousands)	No. of FPGAs	Speedup over software
Binary heap	bheap15	32	15	29	20	1.26
	bheap255	32	255	833	320	2.21
Bubble sort	bubble64	32	64	142	64	7
	bubble512	32	512	1,394	320	25
DES encryption	des4	64	4	47	41	7
	des96	64	96	1,305	320	82
Integer fast Fourier transforms	fft4	3	4	4	17	9
	fft32	17	32	217	64	59
Jacobi	jacobi16×16	8	256	106	85	230
	jacobi32×64	8	2,048	1,126	379	1,562
Conway's Game of Life	life64×16	1	1,024	229	108	597
	life64×64	1	4,096	971	354	1,758
Integer matrix multiply	matmult4×4	16	16	10	18	90
	matmult16×16	16	256	176	64	183
Merge sort	merge8	32	14	14	24	2.60
	merge256	32	510	596	201	1.62
N queens	nqueens16	1	16	14	17	3.96
	nqueens64	1	64	463	215	7
Single-source shortest path	ssp16	16	16	44	14	10
	ssp256	16	256	814	261	52
Multiplicative shortest path	spm16	16	16	156	36	14
	spm32	16	32	310	90	25
Transitive closure	tc512	1	512	187	48	398

logic[3,4] to produce binary code for the FPGA hardware. This binary code consists of individual state machines that represent programs for both the computation threads and statically scheduled communications.

We tested RawLogic on a benchmark suite of 12 general-purpose programs. Executing these benchmarks on the 25-MHz RawLogic prototype achieves 10 to 1,000 speedup over a commercial Sparc 20/71. Table 1 compares execution results to an all-software version executing on a 2.82 SPECint95 SparcStation processor. We also list the number of gates and Xilinx 4013 FPGAs required. By using different problem sizes, we generated cases for each benchmark ranging in size from a few to hundreds of FPGAs. We compiled most of the smaller benchmarks down to configuration binary code and ran them on RawLogic. Because the FPGA compilation step is expensive (several hours per board using 10 workstations), for the larger benchmarks, we report accurate estimates of execution speed provided by our emulation software.

We also attempted to understand the various sources of speedup in each application. For example, Life obtained a 32X speedup over the SparcStation due to bit-level operation optimization (multigranularity), 32X from parallelism, 22X from configurability (a computation on eight bits from eight distinct neighbors). In addition, Life suffered a 3X slowdown from a slower FPGA clock and 13X from communication overhead. This results in an overall speedup of about 600X. A Raw machine shares the advantages of the FPGA system, but it does not suffer from the same clock speed disadvantages. Because centralizing future systems will be physically impossible, all systems will incur some form of communication penalty.

In the near term, Raw architectures will be best suited for stream-based signal-processing computations. In 10 to 15 years, we believe that billion-transistor chip densities, faster switching speeds, and growing compiler sophistication will allow a Raw machine's performance-to-cost ratio to surpass that of traditional architectures for future, general-purpose workloads. Because Raw architectures are field-programmable, we further speculate that they will become a cost-effective alternative to custom hardware in many situations. They thus offer a universal solution for both general- and special-purpose applications.

Using a prototype Raw system, we will attempt to discover the greatest amount of parallelism available at compilation. For programs that the compiler cannot analyze, we plan to implement a system similar to the Multiscalar,[7] but with software-based runtime checks. Because the compiler has access to the underlying hardware mechanisms, we believe that many of these checks can be eliminated.

We now have an operational simulator for a Raw machine and plan a VLSI implementation of Raw with a compiler based on Stanford University's SUIF system. ❖

Acknowledgments

We thank Doug Burger for his incisive technical feedback and his help in condensing this article. This project is funded by US Defense Advanced Research Projects Agency contract DABT63-96-C-0036 and a National Science Foundation Presidential Young Investigator Award. Ikos Systems donated the VirtuaLogic emulation system.

References

1. D.J. Scales, K. Gharachorloo, and C.A. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems VII*, ACM Press, New York, 1996, pp. 174-185.
2. V. Sarkar and J.L. Hennessy, "Compile-Time Partitioning and Scheduling of Parallel Programs," *Proc. ACM SIGPLAN '86 Symp. Compiler Construction,* ACM Press, New York, 1986, pp. 17-26.
3. J. Babb et al., "Logic Emulation with Virtual Wires," *IEEE Trans. CAD*, 1997, to appear.
4. C. Selvidge et al., "TIERS: Topology Independent Pipelined Routing and Scheduling for VirtualWire Compilation," *Proc. ACM Int'l Workshop on FPGAs*, ACM Press, New York, 1995, pp. 12-14.
5. A.V. Aho, M. Ganapathi, and S.W.K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM Trans. Programming Languages and Systems*, Oct. 1989, pp. 491-516.
6. S. Hinrichs et al., "An Architecture for Optimal All-to-All Personalized Communication," *Proc. Sixth Ann. ACM Symp. Parallel Algorithms and Architectures*, ACM Press, New York, 1994, pp. 310-319.
7. G.S. Sohi, S. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 414-425.

Elliot Waingold *is an undergraduate researcher at MIT's Laboratory for Computer Science. He is pursuing a BS and MEng in computer science at MIT.*

Michael Taylor *is an MIT graduate student. He received an AB in computer science from Dartmouth College.*

Devabhaktuni Srikrishna *is a graduate student in electrical engineering and computer science at MIT. He received a BS in mathematics from the California Institute of Technology and is a member of the Mathematical Association of America.*

Vivek Sarkar *is a visiting associate professor at the Laboratory for Computer Science, a senior technical staff member at the IBM Software Solutions Division, and a member of the IBM Academy of Technology. He received a PhD in electrical engineering from Stanford University.*

Walter Lee *is a graduate student at the Laboratory for Computer Science. He received a BS in computer science and an MEng in electrical engineering and computer science, both from MIT.*

Victor Lee *is a senior computer architect for Intel Corp. He received a BS in electrical engineering from the University of Washington and an SM in electrical engineering and computer science from MIT.*

Jang Kim *is a systems analyst at Morgan Stanley. He received a BS and an MEng in computer science from MIT.*

Matthew Frank *is a graduate student at the Laboratory for Computer Science. He received a BS in computer science and mathematics from the University of Wisconsin-Madison.*

Peter Finch *is an undergraduate researcher at the Laboratory for Computer Science. He is pursuing a BS and MEng in computer science at MIT.*

Rajeev Barua *is a PhD candidate in computer science at the Laboratory for Computer Science. He received a BS in computer science and engineering from the Indian Institute of Technology, New Delhi, and an MS in computer science from MIT.*

Jonathan Babb *is a PhD candidate in electrical engineering at MIT and a founder of Virtual Machine Works Inc. He received a BS from the Georgia Institute of Technology and an SM from MIT, both in electrical engineering.*

Saman Amarasinghe *is an assistant professor in the Laboratory for Computer Science. Amarasinghe has a BS in electrical engineering and computer science from Cornell University and an MS and PhD in electrical engineering from Stanford University.*

Anant Agarwal *is an associate professor of electrical engineering and computer science at MIT and a founder of Virtual Machine Works Inc. Agarwal received a BTech in electrical engineering from the Indian Institute of Technology, Madras, and a PhD in electrical engineering from Stanford University.*

Contact the authors at MIT Laboratory for Computer Science, 545 Technology Sq., Rm. 627, Cambridge, MA 02139; http://www.cag.lcs.mit.edu/raw.