Distributed Microarchitectural Protocols in the TRIPS Prototype Processor

Karthikeyan Sankaralingam   Ramadass Nagarajan   Robert McDonald
Rajagopalan Desikan†   Saurabh Drolia   M.S. Govindan   Paul Gratz†   Divya Gulati
Heather Hanson†   Changkyu Kim   Haiming Liu   Nitya Ranganathan
Simha Sethumadhavan   Sadia Sharif†   Premkishore Shivakumar
Stephen W. Keckler   Doug Burger

Department of Computer Sciences   †Department of Electrical and Computer Engineering
The University of Texas at Austin
cart@cs.utexas.edu   www.cs.utexas.edu/users/cart

Abstract

Growing on-chip wire delays will cause many future microarchitectures to be distributed, in which hardware resources within a single processor become nodes on one or more switched micronetworks. Since large processor cores will require multiple clock cycles to traverse, control must be distributed, not centralized. This paper describes the control protocols in the TRIPS processor, a distributed, tiled microarchitecture that supports dynamic execution. It details each of the five types of reused tiles that compose the processor, the control and data networks that connect them, and the distributed microarchitectural protocols that implement instruction fetch, execution, flush, and commit. We also describe the physical design issues that arose when implementing the microarchitecture in a 170M transistor, 130nm ASIC prototype chip composed of two 16-wide issue distributed processor cores and a distributed 1MB nonuniform (NUCA) on-chip memory system.

1 Introduction

Growing on-chip wire delays, coupled with complexity and power limitations, have placed severe constraints on the issue-width scaling of centralized superscalar architectures. Future wide-issue processors are likely to be *tiled* [23], meaning composed of multiple replicated, communicating design blocks. Because of multi-cycle communication delays across these large processors, control must be distributed across the tiles.

For large processors, routing control and data among the tiles can be implemented with microarchitectural networks (or *micronets*). Micronets provide high-bandwidth, flow-controlled transport for control and/or data in a wire-dominated processor by connecting the multiple tiles, which

are clients on one or more micronets. Higher-level microarchitectural protocols direct global control across the micronets and tiles in a manner invisible to software.

In this paper, we describe the tile partitioning, micronet connectivity, and distributed protocols that provide global services in the TRIPS processor, including distributed fetch, execution, flush, and commit. Prior papers have described this approach to exploiting parallelism as well as high-level performance results [15, 3], but have not described the inter-tile connectivity or protocols. Tiled architectures such as RAW [23] use static orchestration to manage global operations, but in a dynamically scheduled, distributed architecture such as TRIPS, hardware protocols are required to provide the necessary functionality across the processor.

To understand the design complexity, timing, area, and performance issues of this dynamic tiled approach, we implemented the TRIPS design in a 170M transistor, 130 nm ASIC chip. This prototype chip contains two processor cores, each of which implements an EDGE instruction set architecture [3], is up to 4-way multithreaded, and can execute a peak of 16 instructions per cycle. Each processor core contains 5 types of tiles communicating across 7 micronets: one for data, one for instructions, and five for control used to orchestrate distributed execution. TRIPS prototype tiles range in size from $1\text{-}9mm^2$. Four of the principal processor elements (instruction and data caches, register files, and execution units) are each subdivided into replicated copies of their respective tile type–for example, the instruction cache is composed of 5 instruction cache tiles, while the computation core is composed of 16 execution tiles.

The tiles are sized to be small enough so that wire delay within the tile is less than one cycle, so can largely be ignored from a global perspective. Each tile interacts only with its immediate neighbors through the various micronets, which have roles such as transmitting operands between

instructions, distributing instructions from the instruction cache tiles to the execution tiles, or communicating control messages from the program sequencer. By avoiding any global wires or broadcast busses–other than the clock, reset tree, and interrupt signals–this design is inherently scalable to smaller processes, and is less vulnerable to wire delays than conventional designs. Preliminary performance results on the prototype architecture using a cycle-accurate simulator show that compiled code outperforms an Alpha 21264 on half of the benchmarks; and we expect these results to improve as the TRIPS compiler and optimizations are tuned. Hand optimization of the benchmarks produces IPCs ranging from 1.5–6.5 and performance relative to Alpha of 0.6–8.

2 ISA Support for Distributed Execution

Explicit Data Graph Execution (EDGE) architectures were conceived with the goal of high-performance, single-threaded, concurrent but distributed execution, by allowing compiler-generated dataflow graphs to be mapped to an execution substrate by the microarchitecture. The two defining features of an EDGE ISA are block-atomic execution and direct communication of instructions within a block, which together enable efficient dataflow-like execution.

The TRIPS ISA is an example of an EDGE architecture, which aggregates up to 128 instructions into a single block that obeys the block-atomic execution model, in which a block is logically fetched, executed, and committed as a single entity. This model amortizes the per-instruction bookkeeping over a large number of instructions and reduces the number of branch predictions and register file accesses. Furthermore, this model reduces the frequency at which control decisions about what to execute must be made (such as fetch or commit), providing the additional latency tolerance to make more distributed execution practical.

2.1 Partitioning TRIPS Blocks

The compiler constructs TRIPS blocks and assigns each instruction to a location within the block. Each block is divided into between two and five 128-byte chunks by the microarchitecture. Every block includes a header chunk which encodes up to 32 read and up to 32 write instructions that access the 128 architectural registers. The read instructions pull values out of the registers and send them to compute instructions in the block, whereas the write instructions return outputs from the block to the specified architectural registers. In the TRIPS microarchitecture, each of the 32 read and write instructions are distributed across the four register banks, as described in the next section.

The header chunk also holds three types of control state for the block: a 32-bit "store mask" that indicates which of
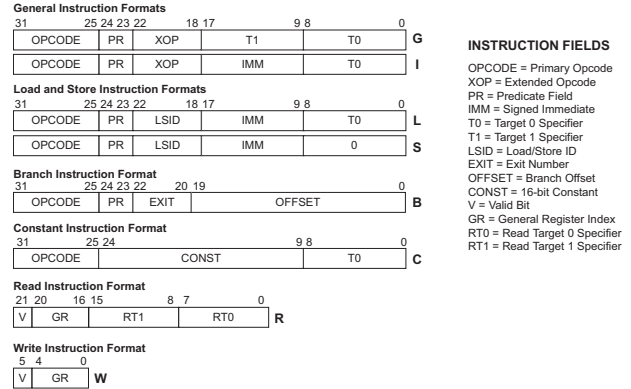

Figure 1. TRIPS Instruction Formats.

the possible 32 memory instructions are stores, block execution flags that indicate the execution mode of the block, and the number of instruction "body" chunks in the block. The store mask is used, as described in Section 4, to enable distributed detection of block completion.

A block may contain up to four body chunks–each consisting of 32 instructions–for a maximum of 128 instructions, at most 32 of which can be loads and stores. All possible executions of a given block must emit the same number outputs (stores, register writes, and one branch) regardless of the predicated path taken through the block. This constraint is necessary to detect block completion on the distributed substrate. The compiler generates blocks that conform to these constraints [19].

2.2 Distributed Instruction Placement

With *direct instruction communication*, instructions in a block send their results directly to intra-block, dependent consumers in a dataflow fashion. This model supports distributed execution by eliminating the need for any intervening shared, centralized structures (e.g. an issue window or register file) between intra-block producers and consumers.

Figure 1 shows that the TRIPS ISA supports direct instruction communication by encoding the consumers of an instruction's result as targets within the producing instruction. The microarchitecture can thus determine precisely where the consumer resides and forward a producer's result directly to its target instruction(s). The nine-bit target fields (T0 and T1) each specify the target instruction with seven bits and the operand type (left, right, predicate) with the remaining two. A microarchitecture supporting this ISA maps each of a block's 128 instructions to particular coordinates, thereby determining the distributed flow of operands along the block's dataflow graph. An instruction's coordinates are implicitly determined by its position its chunk. Other non-traditional elements of this ISA include the "PR" field, which specifies an instruction's predicate and the "LSID" field, which specifies relative load/store ordering.
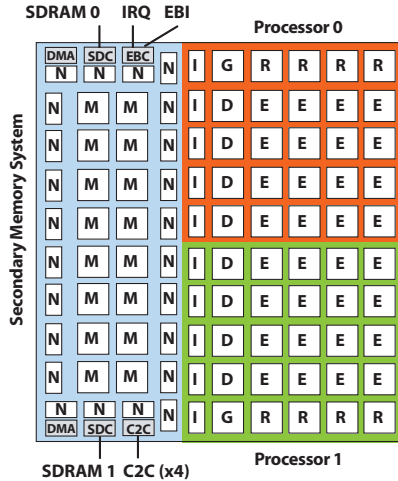
Figure 2. TRIPS prototype block diagram.

Figure 3. TRIPS micronetworks.

3 A Distributed Microarchitecture

The goal of the TRIPS microarchitecture is a processor that is scalable and distributed, meaning that it has no global wires, is built from small set of reused components on routed networks, and can be extended to a wider-issue implementation without recompiling source code or changing the ISA. Figure 2 shows the tile-level block diagram of the TRIPS prototype. The three major components on the chip are two processors and the secondary memory system, each connected internally by one or more micronetworks.

Each of the processor cores is implemented using five unique tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The major processor core micronetwork is the operand network (or OPN), shown in Figure 3. It connects all of the tiles except for the ITs in a two-dimensional, wormhole-routed, 5x5 mesh topology. The OPN has separate control and data channels, and can deliver one 64-bit data operand per link per cycle. A control header packet is launched one cycle in advance of the data payload packet to accelerate wakeup and select for bypassed operands that traverse the network.

Each processor core contains six other micronetworks, one for instruction dispatch (the global dispatch network, or GDN), and five for control: global control network (GCN), for committing and flushing blocks; global status network (GSN), for transmitting information about block completion; global refill network (GRN), for I-cache miss refills; data status network (DSN), for communicating store completion information; and external store network (ESN), for determining store completion in the L2 cache or memory. Links in each of these networks connect only nearest neighbor tiles and messages traverse one tile per cycle. Figure 3 shows the links for four of these networks.

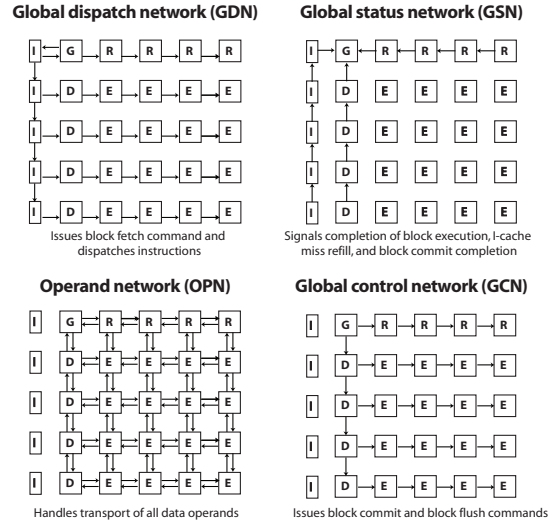This type of tiled microarchitecture is *composable* at de-

sign time, permitting different numbers and topologies of tiles in new implementations with only moderate changes to the tile logic, and no changes to the software model. The particular arrangement of tiles in the prototype produces a core with 16-wide out-of-order issue, 64KB of L1 instruction cache, 32KB of L1 data cache, and 4 SMT threads. The microarchitecture supports up to eight TRIPS blocks in flight simultaneously, seven of them speculative if a single thread is running, or two blocks per thread if four threads are running. The eight 128-instruction blocks provide an in-flight window of 1,024 instructions.

The two processors can communicate through the secondary memory system, in which the On-Chip Network (OCN) is embedded. The OCN is a 4x10, wormhole-routed mesh network, with 16-byte data links and four virtual channels. This network is optimized for cache-line sized transfers, although other request sizes are supported for operations like loads and stores to uncacheable pages. The OCN acts as the transport fabric for all inter-processor, L2 cache, DRAM, I/O, and DMA traffic.

3.1 Global Control Tile (GT)

Figure 4a shows the contents of the GT, which include the blocks' PCs, the instruction cache tag arrays, the I-TLB, and the next-block predictor. The GT handles TRIPS block management, including prediction, fetch, dispatch, completion detection, flush (on mispredictions and interrupts), and commit. It also holds control registers that configure the processor into different speculation, execution, and threading modes. Thus the GT interacts with all of the control networks and the OPN, to provide access to the block PCs.

The GT also maintains the state of all eight in-flight blocks. When one of the block slots is free, the GT accesses the block predictor, which takes three cycles, and emits the
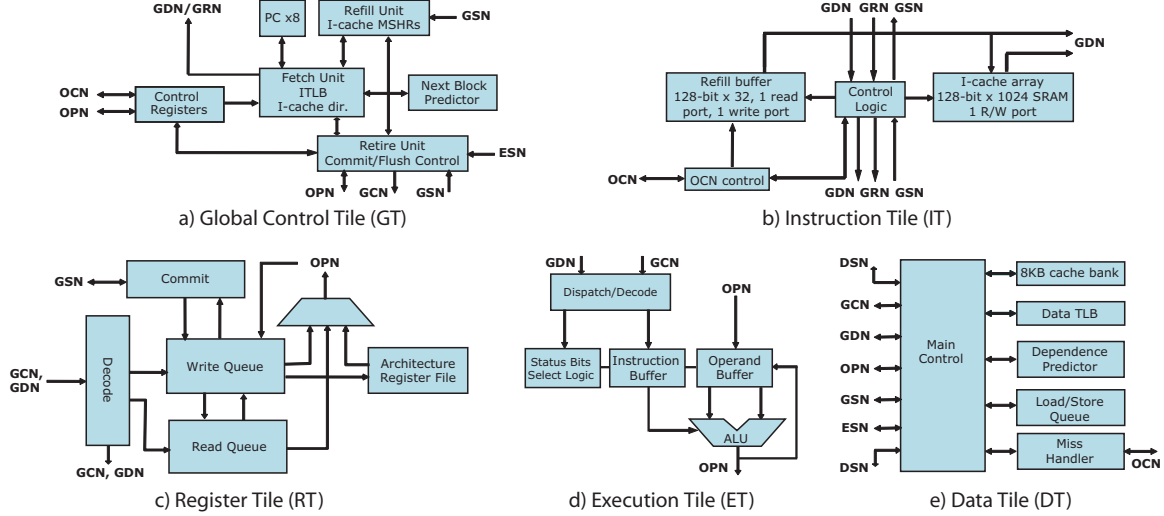
a) Global Control Tile (GT)

b) Instruction Tile (IT)

c) Register Tile (RT)

d) Execution Tile (ET)

e) Data Tile (DT)

Figure 4. TRIPS Tile-level Diagrams.

predicted address of the next target block. Each block may emit only one "exit" branch, even though it may contain several predicated branches. The block predictor uses a branch instruction's three-bit exit field to construct exit histories instead of using taken/not-taken bits. The predictor has two major parts: an exit predictor and a target predictor. The predictor uses exit histories to predict one of eight possible block exits, employing a tournament local/gshare predictor similar to the Alpha 21264 [10] with 9K, 16K, and 12K bits in the local, global, and tournament exit predictors, respectively. The predicted exit number is combined with the current block address to access the target predictor for the next-block address. The target predictor contains four major structures: a branch target buffer (20K bits), a call target buffer (6K bits), a return address stack (7K bits), and a branch type predictor (12K bits). The BTB predicts targets for branches, the CTB for calls, and the RAS for returns. The branch type predictor selects among the different target predictions (call/return/branch/sequential branch). The distributed fetch protocol necessitates the type predictor; the predictor never sees the actual branch instructions, as they are sent directly from the ITs to the ETs.

3.2 Instruction Tile (IT)

Figure 4b shows an IT, which contains a 2-way, 16KB bank of the total L1 I-cache and acts as a slave to the GT, which holds the single tag array. Each of the five 16KB IT banks can hold a 128-byte chunk (for a total of 640 bytes for a maximum-sized block) for each of 128 distinct blocks.

3.3 Register Tile (RT)

To reduce power consumption and delay, the TRIPS microarchitecture partitions its many registers into banks, with one bank in each RT. The register tiles are nodes on the OPN, allowing the compiler to place critical instructions that read and write from/to a given bank close to that bank. Since many def-use pairs of instructions are converted to intra-block temporaries by the compiler, they never access the register file, thus reducing total register bandwidth requirements by approximately 70%, on average, compared to a RISC or CISC processor. The four distributed banks can thus provide sufficient register bandwidth with a small number of ports; in the TRIPS prototype, each RT bank has two read ports and one write port. Each of the four RTs contains one 32-register bank for each of the four SMT threads that the core supports, for a total of 128 registers per RT and 128 registers per thread across the RTs.

In addition to the four per-thread architecture register file banks, each RT contains a read queue and a write queue, as shown in Figure 4c. These queues hold up to eight read and eight write instructions from the block header for each of the eight blocks in flight, and are used to forward register writes dynamically to subsequent blocks reading from those registers. The read and write queues perform a function equivalent to register renaming for a superscalar physical register file, but were less complex to implement due to the read and write instructions in the TRIPS ISA.

3.4 Execution Tile (ET)

As shown in Figure 4d, each of the 16 ETs consists of a fairly standard single-issue pipeline, a bank of 64 reservation stations, an integer unit, and a floating-point unit. All units are fully pipelined except for the integer divide unit, which takes 24 cycles. The 64 reservation stations hold eight instructions for each of the eight in-flight TRIPS blocks. Each reservation station has fields for two 64-bit data operands and a one-bit predicate.

3.5 Data Tile (DT)

Figure 4e shows a block diagram of a single DT. Each DT is a client on the OPN, and holds one 2-way, 8KB L1 data cache bank, for a total of 32KB across the four DTs. Virtual addresses are interleaved across the DTs at the granularity of a 64-byte cache-line. In addition to the L1 cache bank, each DT contains a copy of the load/store queue (LSQ), a dependence predictor, a one-entry back-side coalescing write buffer, a data TLB, and a MSHR that supports up to 16 requests for up to four outstanding cache lines.

Because the DTs are distributed in the network, we implemented a *memory-side* dependence predictor, closely coupled with each data cache bank [17]. Although loads issue from the ETs, a dependence prediction occurs (in parallel) with the cache access only when the load arrives at the DT. The dependence predictor in each DT uses a 1024-entry bit vector. When an aggressively issued load causes a dependence misprediction (and subsequent pipeline flush), the dependence predictor sets a bit to which the load address hashes. Any load whose predictor entry contains a set bit is stalled until all prior stores have completed. Since there is no way to clear individual bit vector entries in this scheme, the hardware clears the dependence predictor after every 10,000 blocks of execution.

The hardest challenge in designing a distributed data cache was the memory disambiguation hardware. Since the TRIPS ISA restricts each block to 32 maximum issued loads and stores and eight blocks can be in flight at once, up to 256 memory operations may be in flight. However, the mapping of memory operations to DTs is unknown until their effective addresses are computed. Two resultant problems are: (a) determining how to distribute the LSQ among the DTs, and (b) determining when all earlier stores have completed–across all DTs–so that a held-back load can issue.

While neither centralizing the LSQ nor distributing the LSQ capacity across the four DTs were feasible options at the time, we solved the LSQ distribution problem largely by brute force. We replicated four copies of a 256-entry LSQ, one at each DT. This solution is wasteful and not scalable (since the maximum occupancy of all LSQs is 25%), but was the least complex alternative for the prototype. The LSQ can accept one load or store per cycle, forwarding data from earlier stores as necessary. Additional details on the DT design can be found in [17].

3.6 Secondary Memory System

The TRIPS prototype supports a 1MB static NUCA [11] array, organized into 16 memory tiles (MTs), each one of which holds a 4-way, 64KB bank. Each MT also includes an on-chip network (OCN) router and a single-entry MSHR. Each bank may be configured as an L2 cache bank or as a scratch-pad memory, by sending a configuration command across the OCN to a given MT. By aligning the OCN with the DTs, each IT/DT pair has its own private port into the secondary memory system, supporting high bandwidth into the cores for streaming applications. The network tiles (NTs) surrounding the memory system act as translation agents for determining where to route memory system requests. Each of them contains a programmable routing table that determines the destination of each memory system request. By adjusting the mapping functions within the TLBs and the network interface tiles (NTs), a programmer can configure the memory system in a variety of ways including as a single 1MB shared level-2 cache, as two independent 512KB level-2 caches (one per processor), as a 1MB on-chip physical memory (no level-2 cache), or many combinations in between. The other six tiles on a chip's OCN are I/O clients which are described in Section 5.

4 Distributed Microarchitectural Protocols

To enable concurrent, out-of-order execution on this distributed substrate, we implemented traditionally centralized microarchitectural functions, including fetch, execution, flush, and commit, with distributed protocols running across the control and data micronets.

4.1 Block Fetch Protocol

The fetch protocol retrieves a block of 128 TRIPS instructions from the ITs and distributes them into the array of ETs and RTs. In the GT, the block fetch pipeline takes a total of 13 cycles, including three cycles for prediction, one cycle for TLB and instruction cache tag access, and 1 cycle for hit/miss detection. On a cache hit, the GT sends eight pipelined indices out on the Global Dispatch Network (GDN) to the ITs. Prediction and instruction cache tag lookup for the next block is overlapped with the fetch commands of the current block. Running at peak, the machine can issue fetch commands every cycle with no bubbles, beginning a new block fetch every eight cycles.

When an IT receives a block dispatch command from the GT, it accesses its I-cache bank based on the index in the GDN message. In each of the next eight cycles the IT sends four instructions on its outgoing GDN paths to its associated row of ETs and RTs. These instructions are written into the read and write queues of the RTs and the reservation stations in the ETs when they arrive at their respective tiles, and are available to execute as soon as they arrive. Since the fetch commands and fetched instructions are delivered in a pipelined fashion across the ITs, ETs, and RTs, the furthest RT receives its first instruction packet ten cycles and its last packet 17 cycles after the GT issues the first fetch command. While the latency appears high, the pipelining
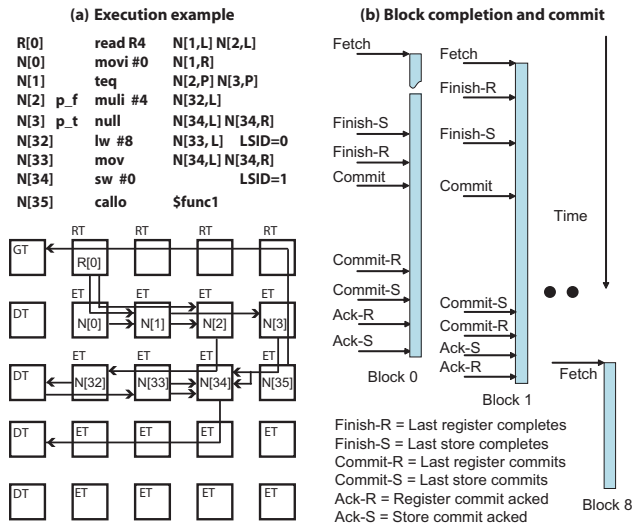
(a) Execution example

R[0]	read R4	N[1,L] N[2,L]
N[0]	movi #0	N[1,R]
N[1]	teq	N[2,P] N[3,P]
N[2] p_f	muli #4	N[32,L]
N[3] p_t	null	N[34,L] N[34,R]
N[32]	lw #8	N[33, L] LSID=0
N[33]	mov	N[34,L] N[34,R]
N[34]	sw #0	LSID=1
N[35]	callo	$func1

(b) Block completion and commit

Figure 5. TRIPS Operational Protocols.

Finish-R = Last register completes
Finish-S = Last store completes
Commit-R = Last register commits
Commit-S = Last store commits
Ack-R = Register commit acked
Ack-S = Store commit acked

enables a high-fetch bandwidth of 16 instructions per cycle in steady state, one instruction per ET per cycle.

On an I-cache miss, the GT instigates a distributed I-cache refill, using the Global Refill Network (GRN) to transmit the refill block's physical address to all of the ITs. Each IT processes the misses for its own chunk independently, and can simultaneously support one outstanding miss for each executing thread (up to four). When the two 64-byte cache lines for an IT's 128-byte block chunk return, and when the IT's south neighbor has finished its fill, the IT signals refill completion northward on the GSN. When the GT receives the refill completion signal from the top IT, the GT may issue a dispatch for that block to all ITs.

4.2   Distributed Execution

An RT may begin to process an arriving read instruction even if the entire block has not yet been fetched. Each RT first searches the write queues of all older in-flight blocks. If no matching, in-flight write to that register is found, the RT simply reads that register from the architectural register file and forwards it to the consumers in the block via the OPN. If a matching write is found, the RT takes one of two actions: if the write instruction has received its value, the RT forwards that value to the read instruction's consumers. If the write instruction is still awaiting its value, the RT buffers the read instruction, which will be woken up by a tag broadcast when the pertinent write's value arrives.

Arriving OPN operands wake up instructions within the ET, which selects and executes enabled instructions. The ET uses the target fields of the selected instruction to determine where to send the resulting operand. Arithmetic operands traverse the OPN to other ETs, while load and store instructions' addresses and data are sent on the OPN to the DTs. Branch instructions deliver their next block addresses to the GT via the OPN.

An issuing instruction may target its own ET or a remote ET. If it targets its local ET, the dependent instruction can be woken up and executed in the next cycle, using a local bypass path to permit back-to-back issue of dependent instructions. If the target is a remote ET, a control packet is formed the cycle before the operation will complete execution, and sent to wake up the dependent instruction early. The OPN is tightly integrated with the wakeup and select logic. When a control packet arrives from the OPN, the targeted instruction is accessed and may be speculatively woken up. The instruction may begin execution in the following cycle as the OPN router injects the arriving operand directly into the ALU. Thus, for each OPN hop between dependent instructions, there will be one extra cycle before the consuming instruction executes.

Figure 5a shows an example of how a code sequence is executed on the RTs, ETs, and DTs. Block execution begins when the read instruction R[0] is issued to RT0, triggering delivery of R4 via the OPN to the left operand of two instructions, `teq` (N[1]) and `muli` (N[2]). When the test instruction receives the register value and the immediate "0" value from the `movi` instruction, it fires and produces a predicate which is routed to the predicate field of N[2]. Since N[2] is predicated on false, if the routed operand has a value of 0, the `muli` will fire, multiply the arriving left operand by four, and send the result to the address field of the `lw` (load word). If the load fires, it sends a request to the pertinent DT, which responds by routing the loaded data to N[33]. The DT uses the load/store IDs (0 for the load and 1 for the store, in this example) to ensure that they execute in the proper program order if they share the same address. The result of the load is fanned out by the `mov` instruction to the address and data fields of the store.

If the predicate's value is 1, N[2] will not inject a result into the OPN, thus suppressing execution of the dependent load. Instead, the `null` instruction fires, targeting the address and data fields of the `sw` (store word). Note that although two instructions are targeting each operand of the store, only one will fire, due to the predicate. When the store is sent to the pertinent DT and the block-ending call instruction is routed to the GT, the block has produced all of its outputs and is ready to commit. Note that if the store is nullified, it does not affect memory, but simply signals the DT that the store has issued. Nullified register writes and stores are used to ensure that the block always produces the same number of outputs for completion detection.

4.3   Block/Pipeline Flush Protocol

Because TRIPS executes blocks speculatively, the pipeline may be flushed periodically, using a distributed

protocol, due to a branch misprediction or a load/store ordering violation. The GT is first notified when a mis-speculation occurs, either by detecting a branch misprediction itself or via a GSN message from a DT indicating a memory-ordering violation. The GT then initiates a flush wave on the GCN which propagates to all of the ETs, DTs, and RTs. The GCN includes a block identifier mask indicating which block or blocks must be flushed. The processor must support multi-block flushing because all speculative blocks after the one that caused the mis-speculation must also be flushed. This wave propagates at one hop per cycle across the array. As soon as it issues the flush command on the GCN, the GT may issue a new dispatch command to start a new block. Because both the GCN and GDN have predictable latencies, the instruction fetch/dispatch command can never catch up with or pass the flush command.

4.4 Block Commit Protocol

Block commit is the most complex of the microarchitectural protocols in TRIPS, since it involves the three phases illustrated in Figure 5b: block completion, block commit, and commit acknowledgment. In phase one, a block is complete when it has produced all of its outputs, the number of which is determined at compile-time and consists of up to 32 register writes, up to 32 stores, and exactly one branch. After the RTs and DTs receive all of the register writes or stores for a given block, they inform the GT using the Global Status Network (GSN). When an RT detects that all block writes have arrived, it informs its east neighbor. The RT completion message is daisy-chained eastward across the RTs, until it reaches the GT indicating that all of the register writes for that block have been received.

Detecting store completion is more difficult since each DT cannot know a priori how many stores will be sent to it. To enable the DTs to detect store completion, we implemented a DT-specific network called the Data Status Network (DSN). Each block header contains a 32-bit *store mask*, which indicates the memory operations (encoded as an LSID bit mask) in the block that are stores. This store mask is sent to all DTs upon block dispatch. When an executed store arrives at a DT, its 5-bit LSID and block ID are sent to the other DTs on the DSN. Each DT then marks that store as received even though it does not know the store's address or data. Thus, a load at a DT learns when all previous stores have been received across all of the DTs. The nearest DT notifies the GT when all of the expected stores of a block have arrived. When the GT receives the GSN signal from the closest RT and DT, and has received one branch for the block from the OPN, the block is complete. Speculative execution may still be occurring within the block, down paths that will eventually be nullified by predicates, but such execution will not affect any block outputs.

During the second phase (block commit), the GT broadcasts a commit command on the Global Control Network and updates the block predictor. The commit command informs all RTs and DTs that they should commit their register writes and stores to architectural state. To prevent this distributed commit from becoming a bottleneck, we designed the logic to support pipelined commit commands. The GT can legally send a commit command on the GCN for a block when a commit command has been sent for all older in-flight blocks, even if the commit commands for the older blocks are still in flight. The pipelined commits are safe because each tile is guaranteed to receive and process them in order. The commit command on the GCN also flushes any speculative in-flight state in the ETs and DTs for that block.

The third phase acknowledges the completion of commit. When an RT or DT has finished committing its architectural state for a given block and has received a commit completion signal from its neighbor on the GSN (similar to block completion detection), it signals commit completion on the GSN. When the GT has received commit completion signals from both the RTs and DTs, it knows that the block is safe to deallocate, because all of the block's outputs have been written to architectural state. When the oldest block has acknowledged commit, the GT initiates a block fetch and dispatch sequence for that block slot.

5 Physical Design/Performance Overheads

The physical design and implementation of the TRIPS chip were driven by the principles of partitioning and replication. The chip floorplan directly corresponds to the logical hierarchy of TRIPS tiles connected only by point-to-point, nearest-neighbor networks. The only exceptions to nearest-neighbor communication are the global reset and interrupt signals, which are latency tolerant and pipelined in multiple stages across the chip.

5.1 Chip Specifications

The TRIPS chip is implemented in the IBM CU-11 ASIC process, which has a drawn feature size of 130nm and seven layers of metal. The chip itself includes more than 170 million transistors in a chip area of 18.30mm by 18.37mm, which is placed in a 47.5mm square ball-grid array package. Figure 6 shows an annotated floorplan diagram of the TRIPS chip taken directly from the design database, as well as a coarse area breakdown by function. The diagram shows the boundaries of the TRIPS tiles, as well as the placement of register and SRAM arrays within each tile. We did not label the network tiles (NTs) that surround the OCN since they are so small. Also, for ease of viewing, we have omitted the individual logic cells from this plot.
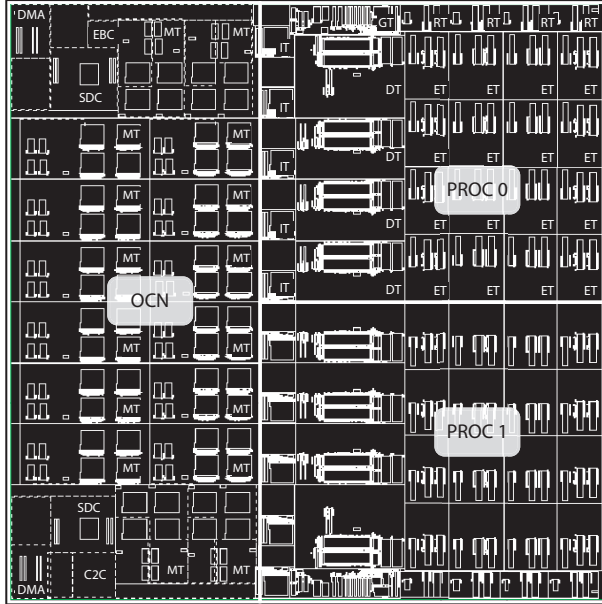
Figure 6. TRIPS physical floorplan.

In addition to the core tiles, TRIPS also includes six controllers that are attached to the rest of the system via the on-chip network (OCN). The two 133/266MHz DDR SDRAM controllers (SDC) each connect to an individual 1GB SDRAM DIMM. The chip-to-chip controller (C2C) extends the on-chip network to a four-port mesh router that gluelessly connects to other TRIPS chips. These links nominally run at one-half the core processor clock and up to 266MHz. The two direct memory access (DMA) controllers can be programmed to transfer data to and from any two regions of the physical address space including addresses mapped to other TRIPS processors. Finally, the external bus controller (EBC) is the interface to a board-level PowerPC control processor. To reduce design complexity, we chose to off-load much of the operating system and run-time control to this PowerPC processor.

TRIPS relies on the trends toward hierarchical design styles with replicated components, but differs from SOCs and CMPs in that the individual tiles are designed to have diverse functions but to cooperate together to implement a more powerful and scalable uniprocessor. The entire TRIPS design is composed of only 11 different types of tiles, greatly simplifying both design and verification. Table 1 shows additional details of the design of each TRIPS tile. The *Cell Count* column shows the number of placeable instances in each tile, which provides a relative estimate of the complexity of the tile. *Array Bits* indicates the total number of bits found in dense register and SRAM arrays on a per-tile basis, while *Size* shows the area of each type of tile. *Tile Count* shows the total number of tile copies across the entire chip, and *% Chip Area* indicates the fraction of the total chip area occupied by that type of tile.

5.2   Area Overheads of Distributed Design

The principal area overheads of the distributed design stem from the wires and logic needed to implement the tile-interconnection control and data networks shown in Table 2. The most expensive in terms of area are the two data networks: the operand network (OPN) and the on-chip network (OCN). In addition to the 141 physical wires per link, the OPN includes routers and buffering at 25 of the 30 processor tiles. The 4-port routers and the eight links per tile consume significant chip area and account for approximately 12% of the total processor area. Strictly speaking, this area is not entirely overhead as it takes the place of the bypass network, which would be much more expensive than the routed OPN for a 16-issue conventional processor. The OCN carries a larger area burden with buffering for four virtual channels at each of the 4-ported routers. It consumes a total of 14% of the total chip area, which is larger than a bus architecture for a smaller scale memory system, but necessary for the TRIPS NUCA cache. In general, the processor control networks themselves do not have a large area impact beyond the cost of the wires interconnecting the tiles. However, we found that full-chip routing was easily accomplished, even with the large number of wires.

Another large source of area overhead due to partitioning comes from the oversized load/store queues in the DT, accounting for 13% of the processor core area. The LSQ cell count and area are skewed somewhat by the LSQ CAM arrays which had to be implemented from discrete latches, because no suitable dense array structure was available in the ASIC design library. Across the entire chip, the area overhead associated with the distributed design stem largely from the on-chip data networks. The control protocol overheads are insignificant, with the exception of the load/store queue.

Table 2. TRIPS Control and Data Networks.

Network	Use	Bits
Global Dispatch (GDN)	I-fetch	205
Global Status (GSN)	Block status	6
Global Control (GCN)	Commit/flush	13
Global Refill (GRN)	I-cache refill	36
Data Status (DSN)	Store completion	72
External Store (ESN)	L1 misses	10
Operand Network (OPN)	Operand routing	141 ($\times$8)
On-chip Network (OCN)	Memory traffic	138 ($\times$8)

5.3 Timing Overheads

The most difficult timing paths we found during logic-level timing optimization were: (1) the local bypass paths from the multi-cycle floating point instructions within the ET, (2) control paths for the cache access state machine in the MT, and (3) remote bypass paths across the operand network within the processor core. The operand network paths are the most problematic, since increasing the latency in cycles would have a significant effect on instruction throughput. In retrospect, we underestimated the latency required for the multiple levels of muxing required to implement the operand router, but believe that a customized design could reduce routing latency. These results indicate a need for further research in ultra-low-latency micronetwork routers.

5.4 Performance Overheads

We examine the performance overheads of the distributed protocols via a simulation-based study using a cycle-level simulator, called *tsim-proc*, that models the hardware at a much more detailed level than higher-level simulators such as SimpleScalar. A performance validation effort showed that performance results from *tsim-proc* were on average within 4% of those obtained from the RTL-level simulator on our test suite and within 10% on randomly generated test programs. We use the methodology of Fields et al. [7] to attribute percentages of the critical path of the program to different microarchitectural activities and partitioning overheads.

The benchmark suite in this study includes a set of microbenchmarks (dct8x8, sha, matrix, vadd), a set of kernels from a signal processing library (cfar, conv, ct, genalg, pm, qr, svd), a subset of the EEMBC suite (a2time01, bezier02, basefp01, rspeed01, tblook01), and a handful of SPEC benchmarks (mcf, parser, bzip2, twolf, and mgrid). In general, these are small programs or program fragments (no more than a few tens of millions of instructions) because we are limited by the speed of *tsim-proc*. The SPEC benchmarks use the reference input set, and we employ subsets of the program as recommended in [18]. These benchmarks reflect what can be run through our simulation environ-

ment, rather than benchmarks selected to leave an unrealistically rosy impression of performance. The TRIPS compiler toolchain takes C or FORTRAN77 code and produces complete TRIPS binaries that will run on the hardware. Although the TRIPS compiler is able to compile major benchmark suites correctly (i.e., EEMBC and SPEC2000) [19], there are many TRIPS-specific optimizations that are pending completion. Until then, performance of compiled code will be lacking because TRIPS blocks will be too small.

While we report the results of the compiled code, we also employed some hand optimization on the microbenchmarks, kernels, and EEMBC programs. We optimized compiler-generated TRIPS high-level assembly code by hand, feeding the result back into the compiler to assign instructions to ALUs and produce an optimized binary. Where possible, we report the results of the TRIPS compiler and the hand-optimized code. We have not optimized any of the SPEC programs by hand and are working to improve compiler code quality to approach that of hand-optimized.

Distributed protocol overheads: To measure the contributions of the different microarchitectural protocols, we computed the critical path of the program and attributed each cycle to one of a number of categories. These categories include instruction distribution delays, operand network latency (including both hops and contention), execution overhead of instructions to fan operands out to multiple target instructions, ALU contention, time spent waiting for the global control tile (GT) to be notified that all of the block outputs (branches, registers, stores) have been produced, and the latency for the block commit protocol to complete. Table 3 shows the overheads as a percentage of the critical path of the program, and the column labeled "Other" includes components of the critical path also found in conventional monolithic cores including ALU execution time, and instruction and data cache misses.

The largest overhead contributor to the critical path is the operand routing, with hop latencies accounting for up to 34% and contention accounting for up to 25%. These overheads are a necessary evil for architectures with distributed execution units, although they can be mitigated through better scheduling to minimize the distance between producers and consumers along the critical path and by increasing the bandwidth of the operand network. For some of the benchmarks, the overheads of replicating and fanning out operand values can be as much as 12%. Most of the rest of the distributed protocol overheads are small, typically summing to less than 10% of the critical path. These results suggest that the overheads of the control networks are largely overlapped with useful instruction execution, but that the data networks could benefit from further optimization.

Total performance: To understand the impact of the distributed protocols on overall performance, we compared execution time on *tsim-proc* to that of a more conventional,

Table 3. Network overheads and preliminary performance of prototype.

| Benchmark | Distributed network overheads as a percentage of program critical path | | | | | | | Preliminary Performance | | | | |
	IFetch	OPN Hops	OPN Cont.	Fanout Ops	Block Complete	Block Commit	Other	Speedup TCC	Hand	IPC Alpha	IPC TCC	IPC Hand
dct8x8	5.39	30.57	10.04	3.76	3.24	2.11	44.89	2.25	2.73	1.69	5.13	4.78
matrix	7.99	20.25	17.24	4.89	4.10	3.17	42.36	1.07	3.36	1.68	2.85	4.12
sha	0.57	17.91	6.29	11.73	0.10	0.66	62.74	0.40	0.91	2.28	1.16	2.10
vadd	7.41	17.66	13.79	5.61	5.99	7.48	42.06	1.46	1.93	3.03	4.62	6.51
cfar	3.75	32.06	9.39	9.78	2.44	0.99	41.59	0.66	0.81	1.53	1.35	1.98
conv	4.10	34.29	16.16	2.71	2.49	2.48	37.77	1.48	2.48	2.08	4.27	5.94
ct	6.23	18.81	16.25	6.04	3.65	3.79	45.23	1.29	3.84	2.31	4.22	5.25
genalg	3.85	18.60	5.76	8.82	2.21	0.62	60.14	0.51	1.46	1.05	1.10	1.65
pm	2.89	25.86	6.21	3.86	1.86	1.03	58.29	0.57	0.99	1.19	1.41	1.96
qr	4.53	22.25	8.85	11.97	2.72	2.23	47.45	0.47	0.98	1.30	1.94	2.36
svd	5.13	15.77	3.84	4.59	3.15	1.46	66.06	0.29	0.68	1.02	1.25	1.11
a2time01	4.94	13.57	6.47	9.52	2.05	4.02	59.43	1.21	4.38	0.95	1.50	4.11
bezier02	2.59	16.92	5.22	12.54	0.21	2.63	59.89	1.61	3.30	1.05	1.91	3.20
basefp01	3.36	13.63	5.44	6.34	2.74	2.90	65.59	1.35	8.02	0.78	1.03	3.55
rspeed01	0.76	28.67	10.61	11.77	0.39	0.14	47.66	1.26	4.18	1.03	1.82	3.38
tblook01	2.88	28.83	9.38	5.68	1.73	0.69	50.81	0.18	0.61	1.44	0.77	1.46
181.mcf	1.64	28.52	6.10	0.00	0.08	0.18	63.48	0.51	—	0.54	0.78	—
197.parser	2.96	30.76	3.99	0.84	0.30	0.66	60.49	0.60	—	1.18	1.10	—
256.bzip2	1.97	33.87	15.17	0.18	0.01	0.18	48.62	0.36	—	1.40	1.27	—
300.twolf	3.01	18.05	3.08	0.75	0.25	0.84	74.02	0.65	—	1.00	1.24	—
172.mgrid	5.06	18.61	25.46	4.03	3.00	2.78	41.06	1.49	—	1.33	4.89	—

albeit clustered, uniprocessor. Our baseline comparison point was a 467MHz Alpha 21264 processor, with all programs compiled using the native Gem compiler with the "-O4 -arch ev6" flags set. We chose the Alpha because it has an aggressive ILP core that still supports low FO4 clock periods, an ISA that lends itself to efficient execution, and a truly amazing compiler that generates extraordinarily high-quality code. We use Sim-Alpha, a simulator validated against the Alpha hardware to take the baseline measurements so that we could normalize the level-2 cache and memory system and allow better comparison of the processor and primary caches between TRIPS and Alpha.

Table 3 shows the performance of the TRIPS processor compared to the Alpha. Since our focus is on the disparity between the processor cores, we simulated a perfect level-2 cache with both processors, to eliminate differences in performance due to the secondary memory system. The first column shows the speedup of TRIPS compiled code (TCC) over the Alpha. We computed speedup by comparing the number of cycles needed to run each program. The second column shows the speedup of the hand-generated TRIPS code over that of Alpha. Columns 3–5 show the instruction throughput (instructions per clock or IPC) of the three configurations. The ratio of these IPCs do not correlate directly to performance, since the instruction sets differ, but they give an approximate depiction of how much concurrency the machine is exploiting. Our results show that on the hand optimized codes, TRIPS executes between 0.6 and 1.8 times

as many instructions as Alpha, largely due to fanout instructions and single-to-double conversions required by TRIPS for codes that use 32-bit floats. The current code bloat is currently larger for compiled code, up to 4 times as many instructions in the worst case.

While these results are far from the best we expect to obtain, they do provide insight into the capabilities of TRIPS. The results show that for the hand optimized programs, the TRIPS distributed microarchitecture is able to sustain reasonable instruction-level concurrency, ranging from 1.1 to 6.5. The speedups over the Alpha core range from 0.6 to just over 8. **sha** sees a slowdown on TRIPS because it is an almost entirely serial benchmark. What little concurrency there is is already mined out by the Alpha core, so the TRIPS processor sees a slight degradation because of the block overheads, such as inter-block register forwarding. Convolution (**conv**), and **vadd** have speedups close to two because the TRIPS core has exactly double the L1 memory bandwidth as Alpha (four ports as opposed to two), resulting in an upper-bound speedup of two. Compiled TRIPS code does not fare as well, but does exceed the performance of Alpha on about half of the benchmarks. The maturation time of a compiler for a new processor is not short, but we anticipate significant improvements as our hyperblock generation and optimization algorithms come on line.

We conclude from this analysis that the TRIPS microarchitecture can sustain good instruction-level concurrency–despite all of the distributed overheads–given kernels with

sufficient concurrency and aggressive handcoding. Whether the core can exploit ILP on full benchmarks, or whether the compiler can generate sufficiently optimized code, remain open questions that are subjects of our current work.

6 Related Work

Much of the TRIPS architecture is inspired by important prior work across many computer architecture domains, including tiled architectures, dataflow architectures, superscalar processors, and VLIW architectures.

Tiled architectures: With transistor counts approaching one billion, tiled architectures are emerging as an approach to manage design complexity. The RAW architecture [23] pioneered research into many of the issues facing tiled architectures, including scalar operand networks, a subset of the class of micronetworks designed for operand transport [22]. Another more recent tiled architecture that, like RAW, uses homogeneous tiles is Smart Memories [14]. Emerging fine-grained CMP architectures, such as Sun's Niagara [12] or IBM's Cell [16], can also be viewed as a tiled architectures. All of these architectures implement one or more complete processors per tile. In general, these other tiled architectures are interconnected at the memory interfaces, although RAW allows register-based inter-processor communication. TRIPS differs in three ways: (1) tiles are heterogeneous, (2) different types of tiles are composed to create a uniprocessor, and (3) TRIPS uses distributed control network protocols to implement functions that would otherwise be centralized in a conventional architecture.

Dataflow architectures: The work most similar to TRIPS are the two recent dataflow-like architectures that support imperative programming languages such as C. These architectures, developed concurrently with TRIPS, are WaveScalar [21] and ASH [2]. WaveScalar breaks programs into blocks (or "waves") similar to TRIPS, but differs in the execution model because all control paths are mapped and executed, instead of the one speculated control path of TRIPS. Other major differences include dynamic, rather than static placement of instructions, no load speculation, and more hierarchy in the networks, since WaveScalar provides many more execution units than TRIPS. ASH uses a similar predication model and dataflow concepts, but targets application-specific hardware for small programs, as opposed to compiling large programs into a sequence of configurations on a programmable substrate like TRIPS. The behavior inside a single TRIPS block builds on the rich history of dataflow architectures including work by Dennis [6], Arvind [1], and hybrid dataflow architectures such as the work of Culler [5] and Iannucci [9].

Superscalar architectures: The TRIPS microarchitecture incorporates many of the high-ILP techniques developed for aggressive superscalar architectures, such as two-level branch prediction and dependence prediction. The TRIPS block atomic execution model is descended from the Block-Structured ISA proposed by Patt et al. to increase the fetch rate for wide issue machines [8]. Other current research efforts also aim to exploit large-window parallelism by means of checkpointing and speculation [4, 20].

VLIW architectures: TRIPS shares some similarities to VLIW architectures in that the TRIPS compiler decides where (but not when) instructions execute. While the TRIPS compiler does not have to decide instruction timing–unlike VLIW architectures–the VLIW compilation algorithms for forming large scheduling regions, such as predicated hyperblocks [13], are also effective techniques for creating large TRIPS blocks.

7 Conclusions

When the first TRIPS paper appeared in 2001 [15], the high-level results seemed promising, but it was unclear (even to us) whether this technology was implementable in practice, or whether it would deliver the performance indicated by the high-level study. The microarchitecture described in this paper is an existence proof that the design challenges unanswered in 2001 were solvable; the distributed protocols we designed to implement the basic microarchitecture functions of instruction fetch, operand delivery, and commit are feasible and do not incur prohibitive overheads. The distributed control overheads are largely overlapped with instruction execution, the logic required to implement the protocols is not significant, and the pipelined protocols are not on critical timing paths.

The data networks, however, carry a larger area and performance burden because they are on the critical paths between data dependent instructions. In the prototype, we are working to reduce these overheads through better scheduling to reduce hop-counts; architectural extensions to TRIPS may include more operand network bandwidth. The original work assumed an ideal, centralized load/store queue, assuming that it could be partitioned in the final design. Because partitioning turned out to be unworkable, we elected to put multiple full-sized copies in every DT, which combined with an area-hungry standard-cell CAM implementation, caused our LSQs to occupy 40% of the DTs. Solving the problem of area-efficiently partitioning LSQs has been a focus of our research for the past year.

These distributed protocols have enabled us to construct a 16-wide, 1024-instruction window, out-of-order processor, which works quite well on a small set of regular, hand-optimized kernels. We have not yet demonstrated that code can be compiled efficiently for this architecture, or that the processor will be competitive even with high-quality code on real applications. Despite having completed the prototype, much work remains in the areas of performance tun-

ing and compilation before we will understand where the microarchitectural, ISA, and compiler bottlenecks are in the design. Once systems are up and running in the fall of 2006, we will commence a detailed evaluation of the capabilities of the TRIPS design to understand the strengths and weaknesses of the system and the technology.

Looking forward, partitioned processors composed of interconnected tiles provide the opportunity to dynamically adjust their granularity. For example, one could subdivide the tiles of a processor to create multiple smaller processors, should the balance between instruction-level and thread-level parallelism change. We expect that such substrates of heterogeneous or homogeneous tiles will provide flexible computing platforms which can be tailored at runtime to match the concurrency needs of different applications.

Acknowledgments

References

[1] Arvind and R. S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[2] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, October 2004.

[3] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[4] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.

[5] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, April 1991.

[6] J. Dennis and D. Misunas. A preliminary architecture for a basic data-flow processor. In *International Symposium on Computer Architecture*, pages 126–132, January 1975.

[7] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[8] E. Hao, P. Chang, M. Evers, and Y. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *International Symposium on Microarchitecture*, pages 191–200, December 1996.

[9] R. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *International Symposium on Computer Architecture*, pages 131–140, May 1988.

[10] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[11] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[12] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[13] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Microarchitecture*, pages 45–54, June 1992.

[14] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *International Symposium on Computer Architecture*, pages 161–171, June 2000.

[15] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *International Symposium on Microarchitecture*, pages 40–51, December 2001.

[16] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *International Solid-State Circuits Conference*, pages 184–185, February 2005.

[17] S. Sethumadhavan, R. McDonald, R. Desikan, D. Burger, and S. W. Keckler. Design and implementation of the TRIPS primary memory system. In *International Conference on Computer Design*, October 2006.

[18] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Technique*, pages 3–14, September 2001.

[19] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, March 2006.

[20] S. Srinivasan, R. Rajwar, H. Akkary, A. Ghandi, and M. Upson. Continual flow pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, October 2004.

[21] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin. Wavescalar. In *36th International Symposium on Microarchitecture*, pages 291–302, December 2003.

[22] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *International Symposium on High Performance Computer Architecture*, pages 341–353, February 2003.

[23] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.