Scaling to the End of Silicon with EDGE Architectures

The TRIPS architecture is the first instantiation of an EDGE instruction set, a new, post-RISC class of instruction set architectures intended to match semiconductor technology evolution over the next decade, scaling to new levels of power efficiency and high performance.

Doug Burger
Stephen W. Keckler
Kathryn S. McKinley
Mike Dahlin
Lizy K. John
Calvin Lin
Charles R. Moore
James Burrill
Robert G. McDonald
William Yoder, and
the TRIPS Team
The University of Texas at Austin

Instruction set architectures have long lifetimes because introducing a new ISA is tremendously disruptive to all aspects of a computer system. However, slowly evolving ISAs eventually become a poor match to the rapidly changing underlying fabrication technology. When that gap eventually grows too large, the benefits gained by renormalizing the architecture to match the underlying technology make the pain of switching ISAs well worthwhile.

Microprocessor designs are on the verge of a post-RISC era in which companies must introduce new ISAs to address the challenges that modern CMOS technologies pose while also exploiting the massive levels of integration now possible. To meet these challenges, we have developed a new class of ISAs, called Explicit Data Graph Execution (EDGE), that will match the characteristics of semiconductor technology over the next decade.

TIME FOR A NEW ARCHITECTURAL MODEL?

The "Architecture Comparisons" sidebar provides a detailed view of how previous architectures evolved to match the driving technology at the time they were defined.

In the 1970s, memory was expensive, so CISC architectures minimized state with dense instruction encoding, variable-length instructions, and small numbers of registers. In the 1980s, the number of devices that could fit on a single chip replaced absolute transistor count as the key limiting resource. With a reduced number of instructions and modes as well as simplified control logic, an entire RISC processor could fit on a single chip. By moving to a register-register architecture, RISC ISAs supported aggressive pipelining and, with careful compiler scheduling, RISC processors attained high performance despite their reduction in complexity.

Since RISC architectures, including the x86 equivalent with µops, are explicitly designed to support pipelining, the unprecedented acceleration of clock rates—40 percent per year for well over a decade—has permitted performance scaling for the past 20 years with only small ISA changes. For example, Intel has scaled its x86 line from 33 MHz in 1990 to 3.4 GHz today—more than a 100-fold increase over 14 years. Approximately half of that increase has come from designing deeper pipelines. However, recent studies show that pipeline scaling is nearly exhausted,[1] indicating that processor design now requires innovations beyond pipeline-centric ISAs. Intel's recent cancellation of its high-frequency Pentium 4 successors is further evidence of this imminent shift.

Future architectures must support four major emerging technology characteristics:

- Pipeline depth limits mean that architects must rely on other fine-grained concurrency mechanisms to improve performance.
- The extreme acceleration of clock speeds has hastened power limits; in each market, future architectures will be constrained to obtain as much performance as possible given a hard

Published by the IEEE Computer Society

Architecture Comparisons

Comparing EDGE with historic architectures illustrates that architectures are never designed in a vacuum—they borrow frequently from previous architectures and can have many similar attributes.

- **VLIW:** A TRIPS block resembles a 3D VLIW instruction, with instructions filling fixed slots in a rigid structure. However, the execution semantics differ markedly. A VLIW instruction is statically scheduled—the compiler guarantees when it will execute in relation to all other instructions.[1] All instructions in a VLIW packet must be independent. The TRIPS processor is a static placement, dynamic issue (SPDI) architecture, whereas a VLIW machine is a static placement, static issue (SPSI) architecture. The static issue model makes VLIW architectures a poor match for highly communication-dominated future technologies. Intel's family of EPIC architectures is a VLIW variant with similar limitations.
- **Superscalar:** An out-of-order RISC or x86 superscalar processor and an EDGE processor traverse similar dataflow graphs. However, the superscalar graph traversal involves following renamed pointers in a centralized issue window, which the hardware constructs—adding instructions individually—at great cost to power and scalability. Despite attempts at partitioned variants,[2] the scheduling scope of superscalar hardware is too constrained to place instructions dynamically and effectively. In our scheduling taxonomy, a superscalar processor is a dynamic placement, dynamic issue (DPDI) machine.
- **CMP:** Many researchers have remarked that a TRIPS-like architecture resembles a chip multiprocessor (CMP) with 16 lightweight processors. In an EDGE architecture, the global control maps irregular blocks of code to all 16 ALUs at once, and the instructions execute at will based on dataflow order. In a 16-tile CMP, a separate program counter exists at each tile, and tiles can communicate only through memory. EDGE architectures are finer-grained than both CMPs and proposed speculatively threaded processors.[3]
- **RAW processors:** At first glance, the TRIPS microarchitecture bears similarities to the RAW architecture. A RAW processor is effectively a 2D, tiled static machine, with shades of a highly partitioned VLIW architecture. The main difference between a RAW processor and the TRIPS architecture is that RAW uses compiler-determined issue order (including the inter-ALU routers), whereas the TRIPS architecture uses dynamic issue, to tolerate variable latencies. However, since the RAW ISA supports direct communication of a producer instruction in one tile to a consuming instruction's ALU on another tile, it could be viewed as a statically scheduled variant of an EDGE architecture, whereas TRIPS is a dynamically scheduled EDGE ISA.
- **Dataflow machines:** Classic dataflow machines, researched heavily at MIT in the 1970s and 1980s,[4] bear considerable resemblances to intrablock execution in EDGE architectures. Dataflow machines were originally targeted at functional programming languages, a suitable match because they have ample concurrency and have side-effect free, "write-once" memory semantics. However, in the context of a contemporary imperative language, a dataflow architecture would need to store and process many unnecessary instructions because of complex control-flow conditions. EDGE architectures use control flow between blocks and support conventional memory semantics within and across blocks, permitting them to run traditional imperative languages such as C or Java, while gaining many of the benefits of more traditional dataflow architectures.
- **Systolic processors:** Systolic arrays are a special historical class of multidimensional array processors[5] that continually process the inputs fed to them. In contrast, EDGE processors issue the set of instructions dynamically in mapped blocks, which makes them considerably more general purpose.

References

1. J.A. Fisher et al., "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proc. 1984 SIGPLAN Symp. Compiler Construction*, ACM Press, 1984, pp. 37-47.
2. K.I. Farkas et al., "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," *Proc. 30th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO-30), IEEE CS Press, 1997, pp. 149-159.
3. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture* (ISCA 95), IEEE CS Press, 1995, pp. 414-425.
4. Arvind, "Data Flow Languages and Architecture," *Proc. 8th Int'l Symp. Computer Architecture* (ISCA 81), IEEE CS Press, 1981, p. 1.
5. H.T. Kung, "Why Systolic Architectures?" *Computer*, Jan. 1982, pp. 37-46.

An EDGE ISA provides a richer interface between the compiler and the microarchitecture.

power ceiling; thus, they must support *power-efficient performance*.
- Increasing resistive delays through global on-chip wires means that future ISAs must be amenable to on-chip *communication-dominated execution*.
- Design and mask costs will make running many application types across a single design desirable; future ISAs should support *polymorphism*—the ability to use their execution and memory units in different ways and modes to run diverse applications.

The ISA in an EDGE architecture supports one main characteristic: *direct instruction communication*. Direct instruction communication means that the hardware delivers a producer instruction's output directly as an input to a consumer instruction, rather than writing it back to a shared namespace, such as a register file. Using this direct communication from producers to consumers, instructions execute in dataflow order, with each instruction firing when its inputs are available. In an EDGE architecture, a producer with multiple consumers would specify each of those consumers explicitly, rather than writing to a single register that multiple consumers read, as in a RISC architecture.

The advantages of EDGE architectures include higher exposed concurrency and more power-efficient execution. An EDGE ISA provides a richer interface between the compiler and the microarchitecture: The ISA directly expresses the dataflow graph that the compiler generates internally, instead of requiring the hardware to rediscover data dependences dynamically at runtime, an inefficient approach that out-of-order RISC and CISC architectures currently take.

Today's out-of-order issue RISC and CISC designs require many inefficient and power-hungry structures, such as per-instruction register renaming, associative issue window searches, complex dynamic schedulers, high-bandwidth branch predictors, large multiported register files, and complex bypass networks. Because an EDGE architecture conveys the compile-time dependence graph through the ISA, the hardware does not need to rebuild that graph at runtime, eliminating the need for most of those power-hungry structures. In addition, direct instruction communication eliminates the majority of a conventional processor's register writes, replacing them with more energy-efficient delivery directly from producing to consuming instructions.

TRIPS: AN EDGE ARCHITECTURE

Just as MIPS was an early example of a RISC ISA, the TRIPS architecture is an instantiation of an EDGE ISA. While other implementations of EDGE ISAs are certainly possible, the TRIPS architecture couples compiler-driven placement of instructions with hardware-determined issue order to obtain high performance with good power efficiency. At the University of Texas at Austin, we are building both a prototype processor and compiler implementing the TRIPS architecture, to address the above four technology-driven challenges as detailed below:

- To *increase concurrency*, the TRIPS ISA includes an array of concurrently executing arithmetic logic units (ALUs) that provide both scalable issue width and scalable instruction window size; for example, increasing the processor's out-of-order issue width from 16 to 32 is trivial.
- To attain *power-efficient high performance*, the architecture amortizes the overheads of sequential, von Neumann semantics over large, 100-plus instruction blocks.
- Since future architectures must be heavily partitioned, the TRIPS architecture uses compile-time instruction placement to *mitigate communication delays,* which minimizes the physical distance that operands for dependent instruction chains must travel across the chip and thus minimizes execution delay.
- Because its underlying dataflow execution model does not presuppose a given application computation pattern, TRIPS offers increased *flexibility*. The architecture includes configurable memory banks, which provide a general-purpose, highly programmable spatial computing substrate. The underlying dataflow-like execution model, in which instructions fire when their operands arrive, is fundamental to computation, efficiently supporting vectors, threads, dependence chains, or other computation patterns as long as the compiler can spatially map the pattern to the underlying execution substrate.

To support conventional languages such as C, C++, or Fortran, the TRIPS architecture uses *block-atomic execution*. In this model, the compiler groups instructions into blocks of instructions, each of which is fetched, executed, and committed atomically, similar to the conventional notion of transactions: A block may either be com-

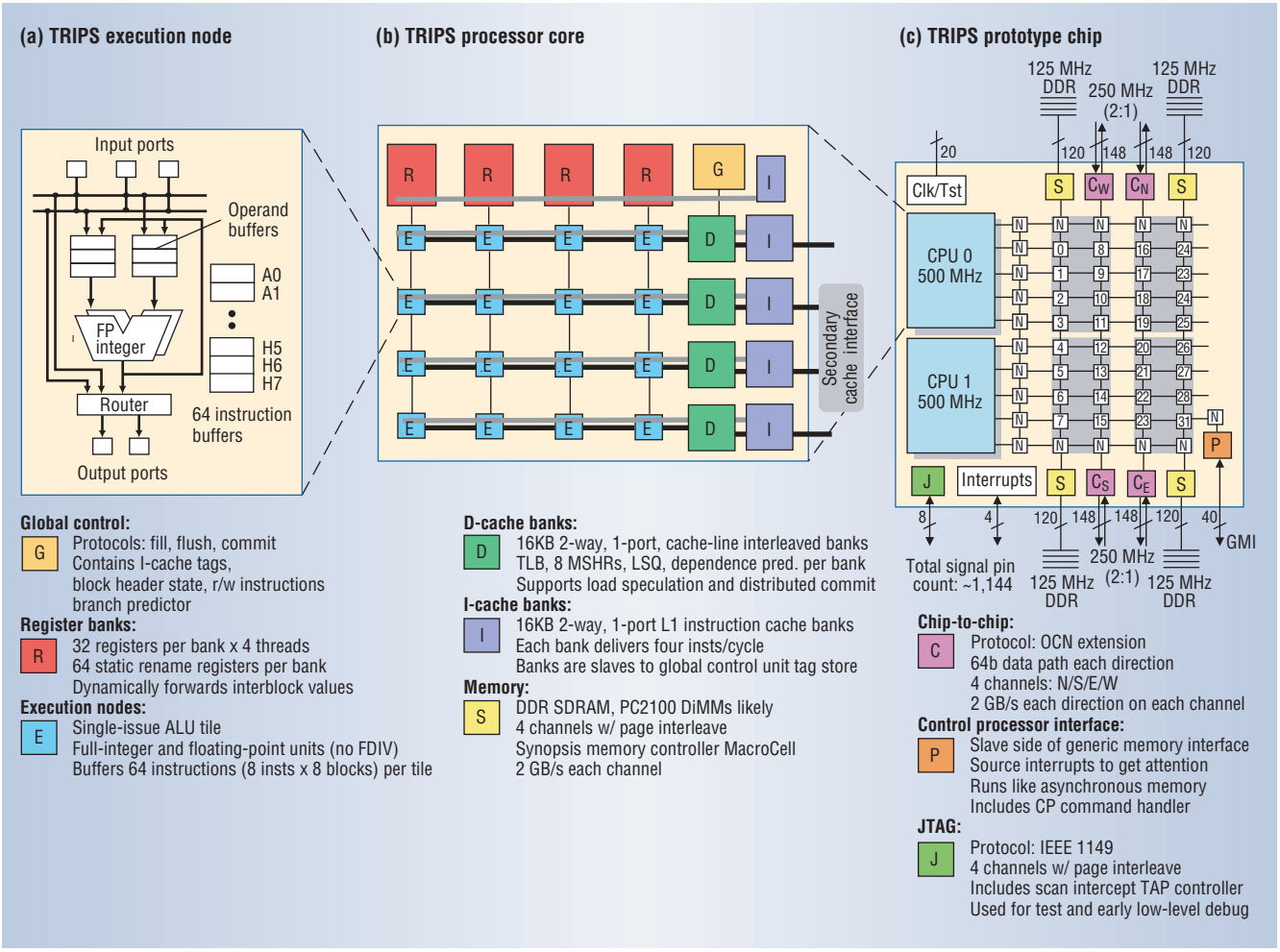

Figure 1. TRIPS prototype microarchitecture. (a) The prototype chip contains two processing cores, each of which is a 16-wide out-of-order issue processor that can support up to 1,024 instructions in flight. Each chip also contains 2 Mbytes of integrated L2 cache, organized as 32 banks connected with a lightweight routing network, as well as external interfaces. (b) The processor core is composed of 16 execution nodes connected by a lightweight network. The compiler builds 128-instruction blocks that are organized into groups of eight instructions per execution node. (c) Each execution node contains a fully functional ALU, 64 instruction buffers, and a router connecting to the lightweight inter-ALU network.

mitted entirely or rolled back; a fraction of a block may not be committed. Each of these TRIPS blocks is a hyperblock[2] that contains up to 128 instructions, which the compiler maps to an array of execution units. The TRIPS microarchitecture behaves like a conventional processor with sequential semantics at the block level, with each block behaving as a "megainstruction." Inside executing blocks, however, the hardware uses a fine-grained dataflow model with direct instruction communication to execute the instructions quickly and efficiently.

TRIPS instructions do not encode their source operands, as in a RISC or CISC architecture. Instead, they produce values and specify where the architecture must route them in the ALU array. For example, a RISC ADD instruction adds the values in R2 and R3, and places the result in R1:

```
ADD R1, R2, R3
```

The equivalent TRIPS instruction specifies only the targets of the add, not the source operands:

```
ADD T1, T2
```

where T1 and T2 are the physical locations (assigned by the compiler) of two instructions dependent on the result of the add, which would have read the result in R1 in the RISC example above. Each instruction thus sends its values to the consumers, and instructions fire as soon as all of their operands arrive.

The compiler statically determines the locations of all instructions, setting the consumer target location fields in each instruction appropriately. Dynamically, the processor microarchitecture fires each instruction as soon as it is ready. When fetching and mapping a block, the processor fetches the instructions in parallel and loads them into the instruction buffers at each ALU in the array. This

TRIPS can achieve power-efficient out-of-order execution across an extremely large instruction window.

block mapping and execution model eliminates the need to go through any fully shared structures, including the register file, for any instruction unless instructions are communicating across distinct blocks. The only exceptions are loads and stores, which must access a bank of the data cache and memory ordering hardware.

To demonstrate the potential of EDGE instruction sets, we are building a full prototype of the TRIPS architecture in silicon, a compiler that produces TRIPS binaries, and a limited runtime system.

Figure 1a is a diagram of the prototype chip, which will be manufactured in 2005 in a 130-nm ASIC process and is expected to run at 500 MHz. The chip contains 2 Mbytes of integrated L2 cache, organized as 32 banks connected with a lightweight routing network. Each of the chip's two processing cores is a 16-wide out-of-order issue processor that can support up to 1,024 instructions in flight, making TRIPS the first kiloinstruction processor to be specified or built.

As Figure 1b shows, each processing core is composed of a 4×4 array of execution nodes connected by a lightweight network. The nodes are not processors; they are ALUs associated with buffers for holding instructions. There are four register file banks along the top, and four instruction and data cache banks along the right-hand side of the core, as well as four ports into the L2 cache network. The compiler builds 128-instruction blocks, organized into groups of eight instructions per node at each of the 16 execution nodes.

To fetch a block of instructions, the global control tile ("G" in Figure 1b) accesses its branch predictor, obtains the predicted block address, and accesses the I-cache tags in the G-tile. If the block's address hits in the I-cache, the G-tile broadcasts the block address to the I-cache banks.

Each bank then streams the block instructions for its respective row into the execution array and into the instruction buffers at each node, shown in Figure 1c. Since branch predictions need occur only once every eight cycles, the TRIPS architecture is effectively unconstrained by predictor or I-cache bandwidth limitations.

Each block specifies register and memory inputs and outputs. Register *read* instructions inject the block inputs into the appropriate nodes in the execution array. Instructions in the block then execute in dataflow order; when the block completes, the control logic writes all register outputs and stores to the register tiles and memory, then it removes the block from the array. Each block emits exactly one branch that produces the location of the subsequent block.

To expose more instruction-level parallelism, the TRIPS microarchitecture supports up to eight blocks executing concurrently. When the G-tile maps a block onto the array, it also predicts the next block, fetching and mapping it as well. In steady state, up to eight blocks can operate concurrently on the TRIPS processor. The renaming logic at the register file bank forwards register values that one block produces directly to consumers in another block to improve performance further.

With this execution model, TRIPS can achieve power-efficient out-of-order execution across an extremely large instruction window because it eliminates many of the power-hungry structures found in traditional RISC implementations. The architecture replaces the associative issue window with architecturally visible instruction buffers constructed from small RAMs. The partitioned register file requires fewer ports because the processor never writes temporary values produced and consumed within a block to the register file. As a result, the TRIPS processor reduces register file accesses and rename table lookups by 70 percent, on average.

The microarchitecture replaces the unscalable broadcast bypass network in superscalar processors with a point-to-point routing network. Although routing networks can be slower than broadcasting, the average distance that operands must travel along the network is short because the compiler places dependent instructions on the same node or on nearby nodes. The instruction cache can provide 16 instructions per cycle and only needs to predict a branch once every eight cycles, reducing the necessary predictor bandwidth. The TRIPS ISA thus amortizes the overheads of out-of-order execution over a 128-instruction block (and with eight blocks, over a 1,024-instruction window), rather than incurring them on every single instruction.

BLOCK COMPILATION

Figure 2 shows an example of how the TRIPS ISA encodes a small, simple compiled block. We assume that the C code snippet in Figure 2a allocates the input integer y to a register. Figure 2b is the same example in MIPS-like assembly code. The variable x is saved in register 2.

The TRIPS compiler constructs the dataflow graph shown in Figure 2c. Two read instructions obtain the register values and forward them to their

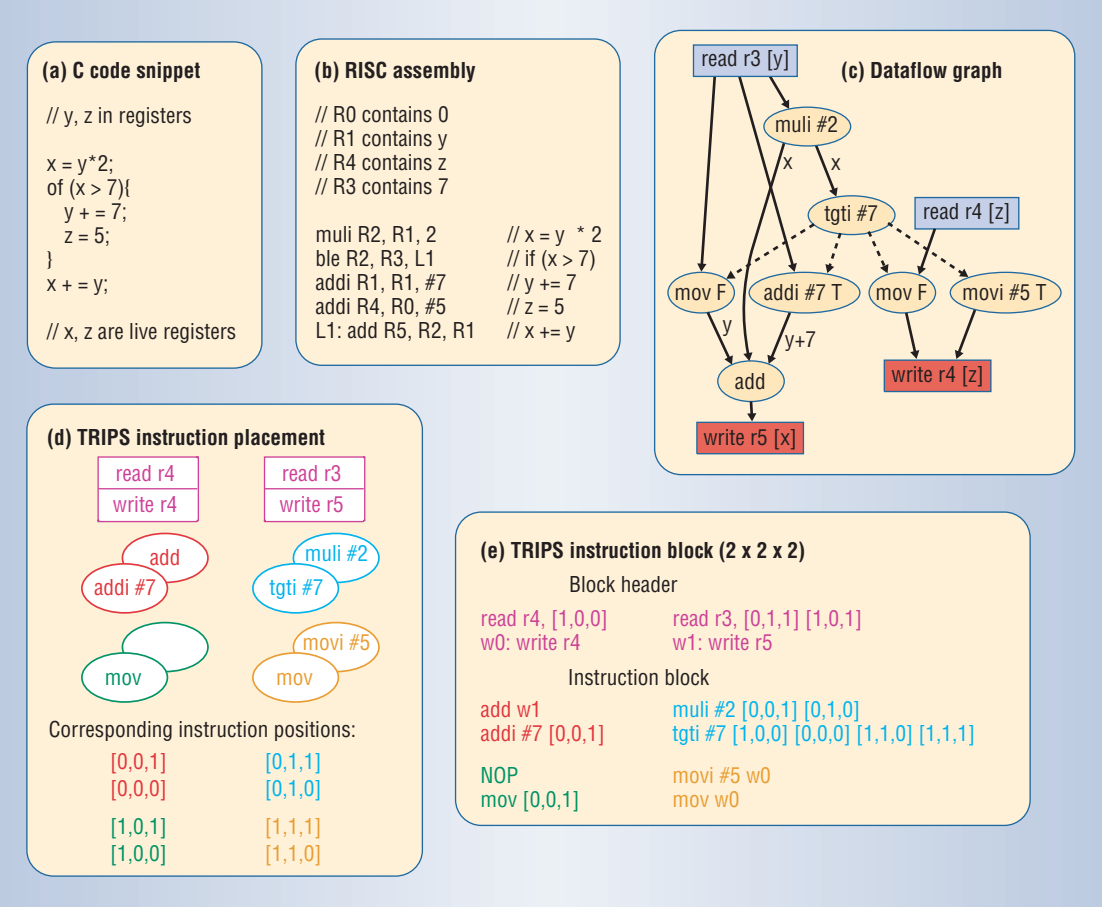

Figure 2. TRIPS code example. (a) In the C code snippet, the compiler allocates the input integer y to a register. (b) In the MIPS-like assembly code, the variable x is saved in register 5. (c) The compiler converts the original branch to a test instruction and uses the result to predicate the control-dependent instructions, which appear as dotted lines in the dataflow graph. (d) Each node in the 2 × 2 execution node array holds up to two buffered instructions. (e) The compiler generates in target form, which correspond to the map of instruction locations at the bottom of part (d).

consuming instructions mapped on the execution array. The compiler converts the original branch to a test instruction and uses the result to predicate the control-dependent instructions, shown as dotted lines. By converting branches to predicates, the compiler creates larger regions with no control flow changes for more effective spatial scheduling.

At runtime, the instructions fire in any order subject to dataflow constraints. The compiled code eventually forwards the block's outputs to the two write instructions and to any other block waiting for those values. Figure 2d shows the dataflow graph (DFG) with the block of instructions scheduled onto a 2 × 2 execution node array, each with an ALU, with up to two instructions buffered per node.

Since the critical path is muli→tgti→addi→add, the compiler assigns the first two instructions to the same ALU so that they can execute with no internode routing delays. No implicit ordering governs the execution of instructions other than the dataflow arcs shown in Figure 2c, so there are no "program order" limitations to instruction issue. When both writes arrive at the register file, the control logic deallocates the block and replaces it with another.

Figure 2e shows the actual code that a TRIPS compiler would generate, which readers can decode using the map of instruction locations at the bottom of Figure 2d. Instructions do not contain their source operands—they contain only the physical locations of their dependent consumers. For example, when the control logic maps the block, the read instruction pulls the value out of register R4 and forwards it to the instruction located in the slot 0 of ALU node [1,1]. As soon as that instruction receives the result of the test condition and the register value, it forwards the value back to register write instruction 0, which then places it back into the register file (R4). If the predicate has a value of false, the instruction doesn't fire.

While this block requires a few extra overhead instructions compared to a RISC ISA, it performs fewer register file accesses—two reads and two writes, as opposed to six reads and five writes. Larger blocks typically have fewer extra overhead instructions, more instruction-level parallelism, and a larger ratio of register file access savings.

COMPILING FOR TRIPS

Architectures work best when the subdivision of labor between the compiler and the microarchitecture matches the strengths and capabilities of each. For future technologies, current execution models strike the wrong balance: RISC relies too little on the compiler, while VLIW relies on it too much.

RISC ISAs require the hardware to discover instruction-level parallelism and data dependences dynamically. While the compiler could convey the

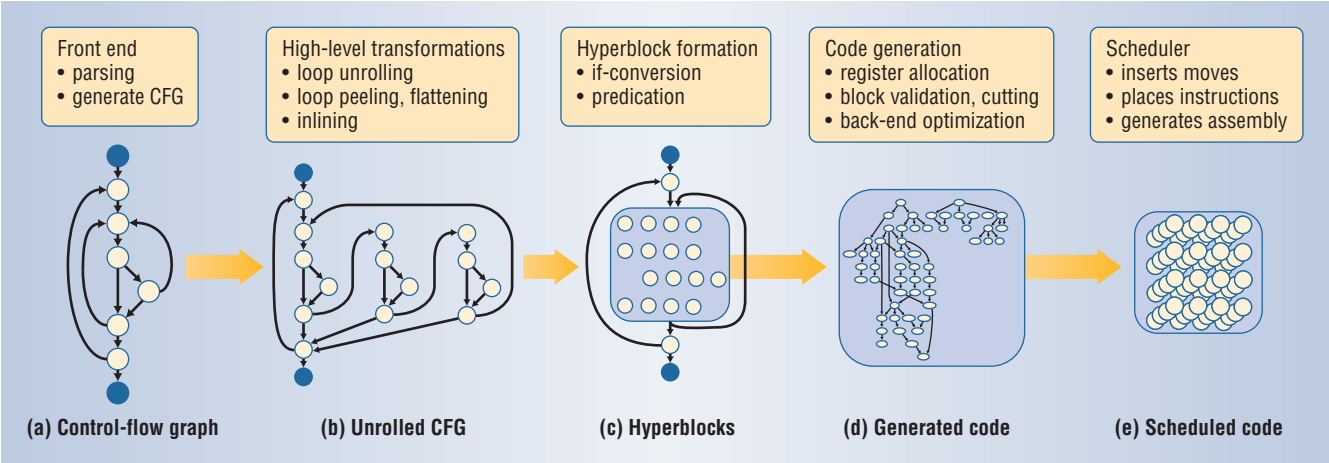

Figure 3. EDGE optimizations in the Scale compiler. (a) A front end parses the code and constructs a control-flow graph (CFG). (b) The CFG after the compiler has unrolled it three times. (c) All of the basic blocks from the unrolled if-converted loop inside a single hyperblock. (d) After inserting the predicates, the compiler generates the code and allocates registers. (e) The scheduler attempts to place independent instructions on different ALUs to increase concurrency and dependent instructions near one another to minimize routing distances and communication delays.

dependences, the ISA cannot express them, forcing out-of-order superscalar architectures to waste energy reconstructing that information at runtime.

VLIW architectures, conversely, put too much of a load on the compiler. They require the compiler to resolve all latencies at compile time to fill instruction issue slots with independent instructions. Since unanticipated runtime latencies cause the machine to stall, the compiler's ability to find independent instructions within its scheduling window determines overall performance. Since branch directions, memory aliasing, and cache misses are unknown at compile time, the compiler cannot generate schedules that best exploit the available parallelism in the face of variable latency instructions such as loads.

For current and future technologies, EDGE architectures and their ISAs provide a proper division between the compiler and architecture, matching their responsibilities to their intrinsic capabilities, and making the job of each simpler and more efficient. Rather than packing together independent instructions like a VLIW machine, which is difficult to scale to wider issue, an EDGE compiler simply expresses the data dependences through the ISA. The hardware's execution model handles dynamic events like variable memory latencies, conditional branches, and the issue order of instructions, *without* needing to reconstruct any compile time information.

An EDGE compiler has two new responsibilities in addition to those of a classic optimizing RISC compiler. The first is forming large blocks (hyperblocks in the TRIPS architecture) that have no internal control flow, thus permitting them to be scheduled as a unit. The second is the spatial scheduling of those blocks, in which the compiler statically assigns instructions in a block to ALUs in the execution array, with the goal of reducing interinstruction communication distances and exposing parallelism.

To demonstrate the tractability of the compiler analyses needed for EDGE architectures, we retargeted the Scale research compiler[3] to generate optimized TRIPS code. Scale is a compilation framework written in Java that was originally designed for extensibility and high performance with RISC architectures such as Alpha and Sparc.

Scale provides classic scalar optimizations and analysis such as constant propagation, loop invariant code motion, dependence analysis, and higher-level transformations such as inlining, loop unrolling, and interchange. To generate high-quality TRIPS binaries, we added transformations to generate large predicated hyperblocks,[2] a new back end to generate *unscheduled* TRIPS code, and a scheduler that maps instructions to ALUs and generates *scheduled* TRIPS assembly in which every instruction is assigned a location on the execution array.

Figure 3 shows the EDGE-specific transformations on the code fragment in Figure 4. Scale uses a front end to parse the code and construct the abstract syntax tree and control-flow graph (CFG), shown in Figure 3a. To form large initial regions, the compiler unrolls loops and inlines functions.

Figure 3b shows the CFG after the compiler has unrolled the inner do-while loop three times. The compiler must take four additional TRIPS-specific steps to generate correct TRIPS binaries.

Large hyperblock formation. The compiler exposes parallelism by creating large predicated hyperblocks, which it can fetch en masse to fill the issue window quickly. Hyperblocks are formally defined as single-entry, multiple-exit regions of code. Because they have internal control-flow transfers, hyperblocks are ideal for mapping onto a spatial substrate. Branches can only jump out of a hyperblock to the start of another hyperblock, never to

```
for (i = 0; i< loopcount; i++) {
  code = 0x1234;
  len = (i % 15) + 1;
  res = 0;
  do {
    res |= code & 1;
    if (res & 0xfddd) res <<= 1;
      code >>= 1,
    } while (--len > 0);
  result += res;
}
```

Figure 4. Modified code fragment from gzip (SPECINT2000).

another point in the same hyperblock or into the middle of another. Large hyperblocks are desirable because they provide a larger region for scheduling and expose more concurrency.

For simplicity, the TRIPS prototype ISA supports fixed 128-instruction hyperblocks, which it fetches at runtime to provide eight instructions at each of the 16 execution nodes. A more complex implementation might permit variable-sized blocks, which map varying numbers of instructions to execution nodes.

To grow hyperblocks, Scale uses inlining, aggressive unrolling (including unrolled while loops), loop peeling/flattening, and if-conversion (a form of predication). Figure 3c shows all the basic blocks from the unrolled and if-converted loop, placed inside a single hyperblock.

At runtime, many hyperblocks will contain useless instructions from CFG nodes included in the hyperblock but not on the dynamically taken path, or from loop iterations unrolled past the end of a loop. To reduce the fraction of unneeded instructions in hyperblocks, Scale uses edge profiling to guide the high-level transformations by identifying both loop counts and infrequently executed basic blocks.

Predicated execution. In previous predication models, evaluating the predicate on every conditional instruction incurs no additional cost. The TRIPS architecture treats a predicate as a dataflow operand that it explicitly routes to waiting predicated consumers. A naive implementation would route a predicate to every instruction in a predicated basic block, creating a wide fan-out problem.

Fortunately, given a dependence chain of instructions that execute on the same predicate, the compiler can choose to predicate only the first instruction in the chain, so the chain does not fire, or only the last instructions in the chain that write registers or memory values. Predicating the first instruction saves power when the predicate is false. Alternatively, predicating only the last, output-producing instructions in a dependence chain generally increases both power and performance by hiding the latency of the predicate computation.

Since the control logic detects block termination when it receives all outputs (stores, register writes, and a branch), in the TRIPS architecture a block must emit the same number of outputs no matter which predicated path is taken. Consequently, the TRIPS ISA supports the notion of *null stores* and *null register writes* that execute whenever a predicated store or register write does not fire. The TRIPS compiler inserts the null assignments optimally using the predicate flow graph, which it generates internally after leaving static single assignment (SSA) form. After inserting the predicates, the compiler generates the code and allocates registers, as shown in Figure 3d, where the hyperblock's dataflow graph is visible.

Generating legal hyperblocks. Scale must confirm that hyperblocks adhere to their legal restrictions before scheduling them onto the execution array. The TRIPS prototype places several restrictions on legal hyperblocks to simplify the hardware: A hyperblock can have no more than 128 instructions, 32 loads or stores along any predicated path, 32 register inputs to the block, and 32 register outputs from the block. These architectural restrictions simplify the hardware at the expense of some hyperblocks being less full to avoid violating those constraints. If the compiler discovers an illegal hyperblock, it splits the block into multiple blocks, allocating intrablock dataflow values and predicates to registers.

Physical placement. After the compiler generates the code, optimizes it, and validates the legality of the hyperblocks, it maps instructions to ALUs and converts instructions to target form, inserting copy operations when a producer must route a value to many consumers. To schedule the code, the compiler assigns instructions to the ALUs on the array, limiting each ALU to at most eight instructions from the block. The scheduler, shown at a high level in Figure 3e, attempts to balance between two competing goals:

- placing independent instructions on different ALUs to increase concurrency, thereby reducing the probability of two instructions competing to issue on the same ALU in the same cycle; and
- placing instructions near one another to minimize routing distances and thus communication delays.

The scheduler additionally exploits its knowledge of the microarchitecture by placing instructions that use the registers near the register banks and by placing critical load instructions near the data cache

A single EDGE architecture can support the three main parallelism classes on the same hardware with the same ISA.

banks. The compiler uses a classic greedy technique to choose the order of instructions to place, with a few additional heuristics to minimize routing distances and ALU contention.

Although the past two years of compiler development have been labor intensive, the fact that we could design and implement this functionality in Scale with a small development team demonstrates the balance in the architecture. The division of responsibilities between the hardware and compiler in the TRIPS architecture is well suited to the compiler's inherent capabilities. Scale can presently compile C and Fortran benchmarks into fully executable TRIPS binaries.

SUPPORTING PARALLELISM

Our work with the TRIPS architecture has shown that a single EDGE architecture can effectively support the three main parallelism classes—instruction, data, and thread—on the same hardware with the same ISA. It may be possible to leverage this inherent flexibility to merge previously distinct markets, such as signal processing and desktop computing, into a single family of architectures that has the same or similar instruction sets and execution models. Area, power, and clock speeds would differentiate implementations to target various power/performance points in distinct markets.

To be fully general and to exploit many types of parallelism, the TRIPS microarchitecture must support the common types of graphs and communication flows across instruction-parallel, data-parallel, and thread-parallel applications without requiring too much specialized hardware support. In previous work, we showed how the TRIPS architecture can harvest instruction-level parallelism from desktop-style codes with complex dependence patterns and, with only minimal additional hardware support, can also support loop-driven data-level parallel codes and threaded parallelism.[4]

Mapping data-level parallelism

Data-level parallel (DLP) applications range from high-end, scientific vector code to graphic processing to low-power, embedded signal processing code. These applications are characterized by frequent, high-iteration loops, large amounts of parallel arithmetic operations, and regular, high-bandwidth access to data sets.

Even though many DLP codes are extremely regular, some such workloads, particularly in the graphics and embedded domains, are becoming less regular, with more complex control flow. EDGE architectures are well suited to both regular and irregular DLP codes because the compiler can map the processing pipelines for multiple data streams to groups of processing elements, using efficient local communication and dataflow-driven activation for execution.

Three additional mechanisms provide significant further benefits to DLP codes. If the compiler can fit a loop body into a single block, or across a small number of blocks, the processor only needs to fetch those instructions once from the I-cache. Subsequent iterations can reuse prior mappings for highly power-efficient loop execution.

In its memory system, the TRIPS prototype has a lightweight network embedded in the cache to support high-bandwidth routing to each of its 32 L2 banks, and it supports dynamic bank configuration using a cache as an explicitly addressable scratchpad. Selectively configuring such memories as scratchpads enables explicit memory management of small on-chip RAMs, similar to signal processing chips such as those in the Texas Instruments C6x series. Our studies show that some DLP codes can benefit directly from higher local memory bandwidth, which developers can add to a TRIPS system by augmenting it with high-bandwidth memory access channels between the processing core and explicitly managed memory. While we will not implement these channels in the prototype, designers may include them in subsequent implementations.

In prior work, we found that adding a small number of additional "universal" hardware mechanisms allowed DLP codes to run effectively on a TRIPS-like processor across many domains such as network processing and cryptography, signal processing, and scientific and graphics workloads.[5] With this additional hardware, a TRIPS-like EDGE processor would, across a set of nine highly varied DLP applications, outperform the best-in-class specialized processor for four applications, come close in two, and fall short in three.

Mapping thread-level parallelism

Most future high-end processors will contain some form of multithreading support. Intel's Pentium 4 and IBM's Power 5 both support simultaneous multithreading (SMT).[6]

We believe that executing a single thread per EDGE processing core will often be more desirable than simultaneously executing multiple threads per core because the EDGE dataflow ISA exposes enough parallelism to effectively use the

core's resources. However, some markets—particularly servers—have copious amounts of thread-level parallelism available, and an EDGE processor should exploit the thread-level parallelism for these applications.

The TRIPS prototype will support simultaneous execution of up to four threads per processing core. The TRIPS multithreading model assigns a subset of the in-flight instruction blocks to each thread. For example, while a single thread might have eight blocks of 128 instructions per block in flight, threads in a four-thread configuration will each have two blocks of 128 instructions in flight.

The processor's control logic moves into and out of multithreading mode by writing to control registers that allocate blocks to threads. In addition to this control functionality, each processor needs a separate copy of an architectural register file for each active thread, as well as some per-thread identifiers augmenting cache tags and other state information.

While the TRIPS prototype maps threads across blocks, thus permitting each thread to have access to all ALUs in the execution array, different EDGE implementations might support other mappings. For example, the hardware could allocate one thread to each column or row of ALUs, thus giving them private access to a register or cache bank. However, these alternative mappings would require more hardware support than the block-based approach used in the TRIPS prototype.

Software challenges

The hardware required to support TLP and DLP applications effectively on an EDGE architecture is but a small increment over mechanisms already present to exploit ILP, making EDGE architectures a good match for exploiting broad classes of parallelism—with high-power efficiency—on a single design. Exploiting an EDGE architecture's flexibility to run heterogeneous workloads comprising a mix of single-threaded, multithreaded, and DLP programs will require modest additional support from libraries and operating systems.

In the TRIPS prototype, memory banks can be configured as caches or as explicitly addressable scratchpads, which lack the process-ID tags that allow processes to share virtually addressed caches. Therefore, to context switch-out/switch-in a process that has activated scratchpad memory, the TRIPS runtime system must save and restore the contents of the scratchpad and reconfigure the memory banks to the appropriate mode of operation. Additionally, to run a heterogeneous mix of jobs on the TRIPS prototype efficiently, the scheduler software should configure processors between single-threaded mode (for most ILP processes and DLP processes) and multithreaded mode (for processes that expose parallelism via large numbers of threads). Further, thread scheduler policies should account for the differing demands of various types of processes.

Overall, because the underlying EDGE substrate provides an abstraction that maps well to a range of different parallelism models, the architecture provides a good balance between hardware and software complexity. Hardware extensions to support ILP, DLP, and TLP are modest, and mechanisms and policies for switching among modes of computation are tractable from a software perspective.

Major ISA changes are traumatic for industry, but solving current challenges may require a major architectural shift.

TO CMP OR NOT TO CMP?

The semiconductor process community has done its job all too well: Computer architects face daunting and interlocking challenges. Reductions in feature size have provided tremendous opportunities by increasing transistor counts, but these advances have introduced new problems of communication delay and power consumption. We believe that finding the solution to these fundamental issues will require a major architecture shift and that EDGE architectures are a good match for meeting these challenges.

Despite the advantages that EDGE architectures offer, major ISA changes are traumatic for industry, especially given the complexity that systems have accrued since the last major shift. However, since then many institutions and companies have developed the technology to incorporate such a new architecture under the hood. For example, Transmeta's code morphing software dynamically translates x86 instructions into VLIW code for its processors. Dynamic translation to an EDGE architecture will likely be simpler than to VLIW, making this technology a promising candidate for solving ISA backward compatibility. We are beginning work on such an effort and have already built a simple PowerPC-to-TRIPS static binary translator.

The competing approach for future systems is explicitly parallel hardware backed by parallelizing software. IBM and Sun Microsystems are both moving to chip multiprocessor (CMP)[7] and chip multithreading models in which each chip contains many processing cores and thread slots that exploit explicitly parallelized threads. Other research efforts, such as Smart Memories[8] and Imagine[9] at

Stanford and RAW[10] at MIT, support DLP workloads with the copious explicit parallelism that software can obtain. This camp argues that future workloads will inevitably shift to be highly parallel and that programmers or compilers will be able to map the parallelism in tomorrow's applications onto a simple, explicitly parallel hardware substrate. Although researchers have consistently made this argument over the past 30 years, the general-purpose market has instead, every time, voted in favor of larger, more powerful uniprocessor cores. The difficulty of scaling out-of-order RISC cores, coupled with IBM's thread-rich server target market, have together driven the emergence of CMPs such as Power 4.

EDGE architectures offer an opportunity to scale the single-processor model further, while still effectively exploiting DLP and TLP when the software can discover it. However, because of the difficulty of discovering and exploiting parallelism, we expect that software will make better use of smaller numbers of more powerful processors. For example, 64 simple processors are much less desirable than four processors, each of which are eight times more powerful than the simple processors. EDGE architectures appear to offer a progressively better solution as technology scales down to the end of silicon, with each generation providing a richer spatial substrate at the expense of increased global communication delays. EDGE ISAs may also be a good match for postsilicon devices, which will likely be communication-dominated as well.

Whether EDGE architectures prove to be a compelling alternative will depend on their performance and power consumption relative to current high-end devices. The prototype TRIPS processor and compiler will help to determine whether this is the case. We expect to have reliable simulation results using optimized compiled code in mid-2004, tape-out in the beginning of 2005, and full chips running in the lab by the end of 2005. ■

Acknowledgments

We thank the following student members of the TRIPS team for their contributions as coauthors of this article: Xia Chen, Rajagopalan Desikan, Saurabh Drolia, Jon Gibson, Madhu Saravana Sibi Govindan, Paul Gratz, Heather Hanson, Changkyu Kim, Sundeep Kumar Kushwaha, Haiming Liu, Ramadass Nagarajan, Nitya Ranganathan, Eric Reeber, Karthikeyan Sankaralingam, Simha Sethumadhavan, Premkishore Sivakumar, and Aaron Smith.

This research is supported by the Defense Advanced Research Projects Agency under contracts F33615-01-C-1892 and F33615-03-C-4106, NSF infrastructure grant EIA-0303609, NSF CAREER grants CCR-9985109 and CCR-9984336, two IBM University Partnership awards, an IBM Shared University Research grant, as well as grants from the Alfred P. Sloan Foundation and the Intel Research Council.

References

1. M.S. Hrishikesh et al., "The Optimal Logic Depth per Pipeline Stage Is 6 to 8 fo4 Inverter Delays," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 14-24.
2. S.A. Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proc. 25th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO-25), IEEE CS Press, 1992, pp. 45-54.
3. R.A. Chowdhury et al., "The Limits of Alias Analysis for Scalar Optimizations," *Proc. ACM SIGPLAN 2004 Conf. Compiler Construction*, ACM Press, 2004, pp. 24-38.
4. K. Sankaralingam et al., "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 422-433.
5. K. Sankaralingam et al., "Universal Mechanisms for Data-Parallel Architectures," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO-36), IEEE CS Press, 2003, pp. 303-314.
6. D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture* (ISCA 95), IEEE CS Press, 1995, pp. 392-403.
7. K. Olukotun et al., "The Case for a Single-Chip Multiprocessor," *Proc. 6th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 94), ACM Press, 1994, pp. 2-11.
8. K. Mai et al., "Smart Memories: A Modular Reconfigurable Architecture," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 161-171.
9. S. Rixner et al., "A Bandwidth-Efficient Architecture for Media Processing," *Proc. 31st Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO-31), IEEE CS Press, 1998, pp. 3-13.
10. E. Waingold et al., "Baring It All to Software: RAW Machines," *Computer*, Sept. 1997, pp. 86-93.

Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, *and* ***Mike Dahlin*** *are associate professors in the Department of Computer Sciences at the University of Texas at Austin. They are members of the ACM and senior members of the IEEE. Contact them at {dburger, skeckler, mckinley, dahlin}@cs.utexas.edu.*

Lizy K. John *is an associate professor in the Department of Electrical and Computer Engineering at the University of Texas at Austin. She is a member of the ACM and a senior member of the IEEE. Contact her at ljohn@ece.utexas.edu.*

Calvin Lin *is an associate professor in the Department of Computer Sciences at the University of Texas at Austin and is a member of the ACM. Contact him at lin@cs.utexas.edu.*

Charles R. Moore*, a senior research fellow at the University of Texas at Austin from 2002 through 2004, is currently a Senior Fellow at Advanced Micro Devices. Contact him at chuck.moore@amd.com.*

James Burrill *is a research fellow in the Computer Science Department at the University of Massachusetts at Amherst. Contact him at burrill@cs.umass.edu.*

Robert G. McDonald *is the chief engineer for the TRIPS prototype chip at the University of Texas at Austin. Contact him at robertmc@cs.utexas.edu.*

William Yoder *is a research programmer at the University of Texas at Austin and a member of the ACM and the IEEE. Contact him at byoder@cs.utexas.edu.*