Compiling for EDGE Architectures *

Aaron Smith Jim Burrill[1] Jon Gibson Bertrand Maher Nick Nethercote
Bill Yoder Doug Burger Kathryn S. McKinley

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

[1]Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Explicit Data Graph Execution (EDGE) architectures offer the possibility of high instruction-level parallelism with energy efficiency. In EDGE architectures, the compiler breaks a program into a sequence of structured blocks that the hardware executes atomically. The instructions within each block communicate directly, instead of communicating through shared registers. The TRIPS EDGE architecture imposes several restrictions on its blocks to simplify the microarchitecture: each TRIPS block has at most 128 instructions, issues at most 32 loads and/or stores, and executes at most 32 register bank reads and 32 writes. To detect block completion, each TRIPS block must produce a constant number of outputs (stores and register writes) and a branch decision.

The goal of the TRIPS compiler is to produce TRIPS blocks full of useful instructions while enforcing these constraints. This paper describes a set of compiler algorithms that meet these sometimes conflicting goals, including an algorithm that assigns load and store identifiers to maximize the number of loads and stores within a block. We demonstrate the correctness of these algorithms in simulation on SPEC2000, EEMBC, and microbenchmarks extracted from SPEC2000 and others. We measure speedup in cycles over an Alpha 21264 on microbenchmarks.

1. Introduction

Technology trends–growing wire delays, power consumption limits, and diminishing clock rate improvements–are presenting historical instruction set architectures such as RISC, CISC, and VLIW with difficult challenges [1]. To show continued performance growth, future microprocessors must exploit concurrency power efficiently. An important question for any future system is the division of responsibilities between programmer, compiler, and hardware to discover and exploit this concurrency.

In previous solutions, CISC processors intentionally placed few ISA-imposed requirements on the compiler to expose concurrency. In-order RISC processors required the compiler to schedule instructions to minimize pipeline bubbles for effective pipelining concurrency. With the advent of large-window out-of-order microarchitectures, however, both RISC and CISC processors rely mostly on the hardware to support superscalar issue. These processors use a *dynamic placement, dynamic issue* execution model that requires the hardware to construct the program dataflow graph on the fly, with little compiler assistance. VLIW processors, conversely, place most of the burden of identifying concurrent instructions on the compiler, which must fill long instruction words at compile time. This *static placement, static issue* execution model works well when all delays are known statically, but in the presence of variable cache and memory latencies, filling wide words has proven to be a difficult challenge for the compiler [20, 24].

Explicit Data Graph Execution (or EDGE) architectures partition the work between the compiler and the hardware differently than do RISC, CISC, or VLIW architectures [1, 10, 30, 22, 28, 32], with the goal of exploiting fine-grained concurrency at high energy efficiency. An EDGE architecture has two distinct features that require new compiler support. First, the compiler is responsible for partitioning the program into a sequence of structured *blocks*, which logically execute atomically. The EDGE ISA defines the structure of, and the restrictions on, these blocks. Second, instructions within each block employ *direct instruction communication*. The compiler encodes instruction dependences explicitly, eliminating the need for the hardware to discover dependences dynamically. For example, if instruction A produces a value for instruction B, A's instruction bits specify B as a consumer, and the hardware routes

*This research is supported by the Defense Advanced Research Projects Agency under contract F33615-03-C-4106 and NSF CISE infrastructure grant EIA-0303609.

A's output operand directly to B.

The TRIPS architecture is an EDGE ISA that employs a hybrid *static placement, dynamic issue* execution model. The TRIPS microarchitecture consists of a four by four grid of execution units with banks of registers at the top, and cache memories to the left, as depicted in Figure 1. This hardware issues instructions dynamically and out-of-order when their source operands become available. In the TRIPS architecture, the compiler is responsible for placing instructions on an execution substrate to minimize physical distances between producing and consuming instructions, and for encoding dependences in the instructions themselves.

This execution paradigm has two potential advantages over traditional, single-instruction-granularity architectures. First, out-of-order execution has the potential to be more power efficient than in RISC/CISC ISAs, since the hardware is not required to derive inter-instruction dependences within a block. Second, executing at the granularity of blocks amortizes the overhead of instruction dispatch and mapping (register file accesses, branch prediction, and instruction cache lookups) over a large number of instructions, reducing both energy consumption and enabling higher instruction-level concurrency.

However, these potential advantages come at the cost of additional responsibilities for the compiler, which are (1) forming dense blocks that obey the structural requirements specified by the ISA, and (2) encoding the dependences in the instructions and placing the instructions to minimize inter-ALU communication latencies. Previous work describes the latter compiler responsibility [27]. This paper focuses on the former–the compiler flow and algorithms necessary to generate effective and legal blocks.

The TRIPS ISA places a number of structural restrictions on blocks to permit simpler hardware, which are more restrictive than those for traditional hyperblocks [24] and superblocks [19]. These restrictions make the compiler's task of forming dense but legal blocks more challenging. Fewer restrictions makes for a simpler compiler but more complicated hardware. The TRIPS ISA and prototype hardware employ four restrictions on blocks intended to strike a balance between software and hardware complexity. They are: (1) fixed block sizes (maximum of 128 instructions), (2) restricted number of loads and stores (no more than 32 may issue per block), (3) restricted register accesses (no more than eight reads and eight writes to each of four banks per block), and (4) constant number of block outputs (each block must always generate a constant number of register writes and stores, plus exactly one branch).

While these restrictions are not difficult to meet for small blocks, the TRIPS compiler's goal is to fill each block as much as possible, maximizing the number of instructions in the dynamic window, while adhering to the ISA-specified block constraints. To increase TRIPS blocks beyond single basic blocks and meet constraints (1) and (2), the compiler extends prior work on predicated hyperblock formation [5, 12, 13, 19, 24]. The compiler uses an algorithm that incrementally combines TRIPS blocks, checking block legality before combination.

The hardware enforces sequential memory semantics by employing a 5-bit load/store ID (LSID) in each load or store instruction to maintain a total order of memory operations. To mitigate the effect of the limit on LSIDs (contraint 2), and to produce a constant number of outputs (constraint 4), the compiler implements an algorithm that reuses LSIDs on control-independent paths, guaranteeing that only one load or store with a given LSID will fire at runtime.

To satisfy the banking constraints (constraint 3), the compiler modifies a linear-scan register allocator [16, 33]. The compiler uses SSA [17] to reveal register writes and stores on disjoint paths, and to guarantee that each path through the TRIPS block produces one and only one write to any register, and one and only one store to a given LSID (constraint 4). If any of these constraints are violated (for example, if the insertion of spill code during register allocation causes a block to exceed its LSID limit or instruction count), the compiler splits the offending block using reverse-if conversion [4], and then re-runs register allocation, iterating as necessary until all TRIPS blocks satisfy the architectural constraints.

We show that with these algorithms, the compiler can correctly compile the major desktop and embedded benchmark suites: SPEC2000 and EEMBC. On a number of microbenchmarks extracted from SPEC2000 and a few other programs, we show that the compiler can generate code that provides a 83% speedup over an Alpha 21264 using gcc with full optimization.

2. TRIPS EDGE Architecture Background

This section provides an overview of the TRIPS EDGE ISA and microarchitecture [10, 22, 27, 28, 30]. EDGE architectures break programs into blocks that are atomic logical units, and within those blocks, instructions directly target other instructions without specifying their source operands in the instruction word. An example of an EDGE instruction takes the following form: `ADD 126(0), 37(p)`. When the ADD instruction receives two operands, it computes the result, and forwards its result to the left-hand (0) operand of instruction number 126 and the predicate field (p) of instruction number 37. Upon receiving its operands and predicate, instruction 37 will fire only if the predicate condition evaluates to true.

Figure 1 shows the TRIPS prototype processor microarchitecture. The execution core consists of an array of 16 ALUs connected by a lightweight switching network. The TRIPS microarchitecture binds each instruction number within a block to a specific reservation station coupled to an ALU. The microarchitecture consists of five types of

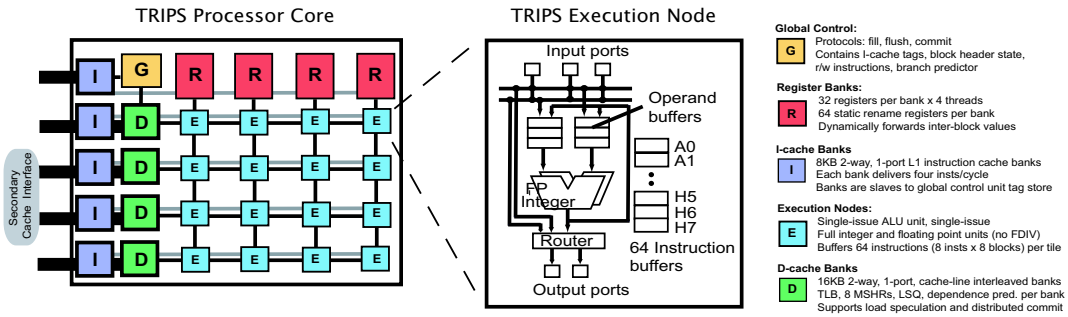

Figure 1. TRIPS Prototype Processor Core

tiles: G-tile (global control), R-tile (registers), E-tile (execution), I-tile (instruction cache), and D-tile (data cache).

2.1. TRIPS Block Execution

The global control tile (G-tile) determines which TRIPS blocks to execute, and tracks up to eight blocks in flight simultaneously. To fetch a block, the G-tile issues a block address prediction, checks its I-cache tags, and upon a hit, signals the slave I-cache banks in the lower four I-tiles to stream 32 instructions each to their respective row of ALUs. The top I-tile contains each block's header, which consists of control information and the register read/write instructions, which inject initial register values into a block and write the block outputs back to the global register file. These instructions are sent to the four R-tiles, which hold buffers for the read and write instructions, one bank each of the architectural register file, and forwarding logic to send writes from earlier blocks to reads in later ones. Each E-tile loads its eight instructions from a block into its reservation stations, thus requiring a total of 64 reservation stations (eight per block times eight blocks). The maximum window size is 1024 instructions: 128 instructions maximum per block times eight blocks in flight.

When a block begins execution, the register read instructions fire (one per bank per cycle) and the interconnect routes their values to their target instructions in the E-tiles, taking one cycle per Manhattan-distance hop. Each E-tile contains an integer and floating-point unit, and can issue one ready instruction per cycle. Dependent instructions on the same E-tile can issue in back-to-back cycles, but there is a one-cycle bubble for each routing hop in the E-tile network if dependent instructions reside in different E-tiles. The instructions in the block fire in dataflow order, with loads and stores routed to the data-cache tiles (D-tiles) on the left of the array. The four D-tiles each hold one cache-line interleaved, 8KB bank of the data cache. A load is either deferred in a D-tile's load/store queue if the D-tile predicts it is dependent on a prior store [15, 26], or speculatively performs a cache lookup and routes its value to consumers in the E-tile array.

When a branch fires, it routes the computed next-block address to the G-tile, which confirms that the block prediction was correct (or rolls back). The block commits when it is the oldest block and receives all of its outputs. Block completion is detected when the D-tiles signal the G-tile that all that block's stores have been received, and when the R-tiles signal similarly for the register writes. The G-tile then committs the block, predicts a new next-block address, and fetches the new block into the just-committed block's slot. On an exception or interrupt, execution rolls back to the last committed block boundary, providing precise exceptions at a block level.

2.2. TRIPS Block Constraints

By organizing instructions into blocks, the TRIPS microarchitecture supports out-of-order execution–of both instructions within blocks and across blocks–without requiring associative tags to compare incoming operands. However, too few restrictions on blocks would require more complicated hardware (e.g., if a block could emit a different number of outputs each time it executed). A long-term goal is to find the right compiler-hardware "sweet spot" in the architectural definition of a TRIPS block; one that permits the compiler to form large, full blocks, without requiring unnecessarily complex hardware. We attempted to find that balance by choosing the following architectural constraints on TRIPS blocks:

Fixed Block Size: All blocks contain at most 128 compute instructions (register reads and writes are additional). This fixed-size restriction permits the architecture to align each instruction cache bank with its reservation stations, and also simplifies the mapping of a block's instructions into hardware resources.

Load/Store Identifiers: Each load and store contains a 5-bit ordering identifier or LSID (e.g., a store with LSID 7 must logically complete before a load with LSID 8 and a store with LSID 9). There may be more than 32 static load/store instructions per block, since loads or stores down disjoint predicate paths may share the same ID, but at most one memory operation with a given LSID may fire, and

there are at most 32 LSIDs.

Register Constraints: Each register bank issues at most eight read and eight write instructions per block. This limit reduces the hardware buffering required for each block. Since the TRIPS hardware can dynamically forward in-flight writes to reads from later blocks, all in-flight writes must be buffered. Each R-tile holds 64 buffered reads and 64 buffered writes (eight per bank per block times eight blocks in flight). Eliminating the banking restriction would increase the buffering at each R-tile four-fold.

Constant Output: To detect that a block is complete, each TRIPS block emits a consistent number of register writes and stores, plus exactly one branch. The hardware observes the number of register writes by counting the write instructions issued to the R-tiles. The compiler encodes the number of stores in a 32-bit store mask in the block header, enabling the D-tiles to detect when it receives all of a block's stores. If the code contains a store or register write down one predicated path but not another, the compiler must insert additional instructions on the alternative path to ensure that the hardware does not wait for a store or write that will never issue. This block restriction adds instruction overhead, but considerably simplifies detection of block termination.

3. Compiler Approach and Algorithms

This section describes both the compiler flow and algorithms needed to generate correct TRIPS blocks. To compile to the TRIPS ISA, we made a number of additions to the Scale retargetable compiler for C and Fortran [37]. Scale is written in Java and produces TRIPS assembly language (TASL) [34], Alpha assembly, SPARC assembly, and C. Figure 2 shows the high-level compiler flow, with the shaded portions of the figure listing the basic phases and intermediate representations. Above each phase the figure also shows an example intermediate representation.

Phase I performs inlining, loop unrolling, and scalar optimizations. Phase II generates code, forms TRIPS blocks, performs register allocation, and splits the blocks if necessary. Phase III performs backend optimizations on the TRIPS blocks, including assigning LSIDs, inserting dummy stores, and applying peephole optimizations. Finally, phase IV lowers the code to TRIPS assembly (TASL [34]), building operand fanout trees and placing instructions to reduce operand routing latencies and exploit parallelism.

3.1. Front-end Transformations

In phase I, Scale first transforms programs into a control-flow graph (CFG), and then performs inlining and unrolling. Scale performs alias analysis on the CFG and uses the results to build a static single assignment (SSA) [17] form of the CFG. Scale then performs the following scalar optimizations on the SSA form: global variable replacement, sparse conditional constant propagation, array access strength reduction, loop invariant code motion, copy propagation, scalar replacement, global value numbering, expression tree height reduction, useless copy removal, and dead variable elimination. After scalar optimizations, Scale converts the CFG out of SSA.

The scalar optimizations produce more efficient code, which improves the TRIPS blocks when they are formed in phase II. However, inlining and loop unrolling both have the potential to exceed block constraints, and will force the block splitter (described in the next subsection) to split the blocks into legal TRIPS blocks. Currently, we use a default unroll factor of three for unrolling. Loops are thus often unrolled imperfectly with respect to the block constraints.

3.2. Code Generation

In phase II, the compiler first generates TIL instructions from the control flow graph. TIL is a RISC-like TRIPS intermediate representation [35] that does not consider physical placement of instructions. We use the code snippet in Figure 3 as a running example of how the compiler transforms a program from C to a legal TRIPS block.

To enable transformations on predicated code, the backend forms a three-level hierarchical graph that builds on prior hyperblock research [4]:

TRIPS Block Flow Graph (TFG): A TFG represents a single procedure. Each node in the TFG corresponds to one TRIPS block, while each edge represents control flow between TRIPS blocks. Each TRIPS block is represented as a *predicate flow graph*.

Predicate Flow Graph (PFG): A PFG node is represented as a *predicate block* and each PFG edge represents control flow between predicate blocks that will be attained using predication. The first node in each PFG is the unpredicated entry into a TRIPS block. To provide a convenient location to place spills and register write instructions, we add an empty, unpredicated, final merge node to every PFG.

Predicate Block: Each predicate block is a basic block of TIL instructions with uniform predication; either no instructions are predicated, or all are predicated on the same predicate.

Next the compiler identifies the TFG nodes to combine into TRIPS blocks through if-conversion [2] and block formation merges these nodes together. All the subsequent phases analyze and transform TFG nodes into legal TRIPS blocks that meet the architectural constraints, and thus differ substantially from previous work on hyperblocks and superblocks [4, 9, 19, 29, 24].

Figures **??**(b) and (c) show TFG nodes as shaded regions. These nodes correspond to TRIPS blocks that do not yet meet all the constraints. Figures 2(b) and (c) also show the details of PFGs in some of the TFG nodes; in these detailed TFG nodes each sub-node is a single predicate block.

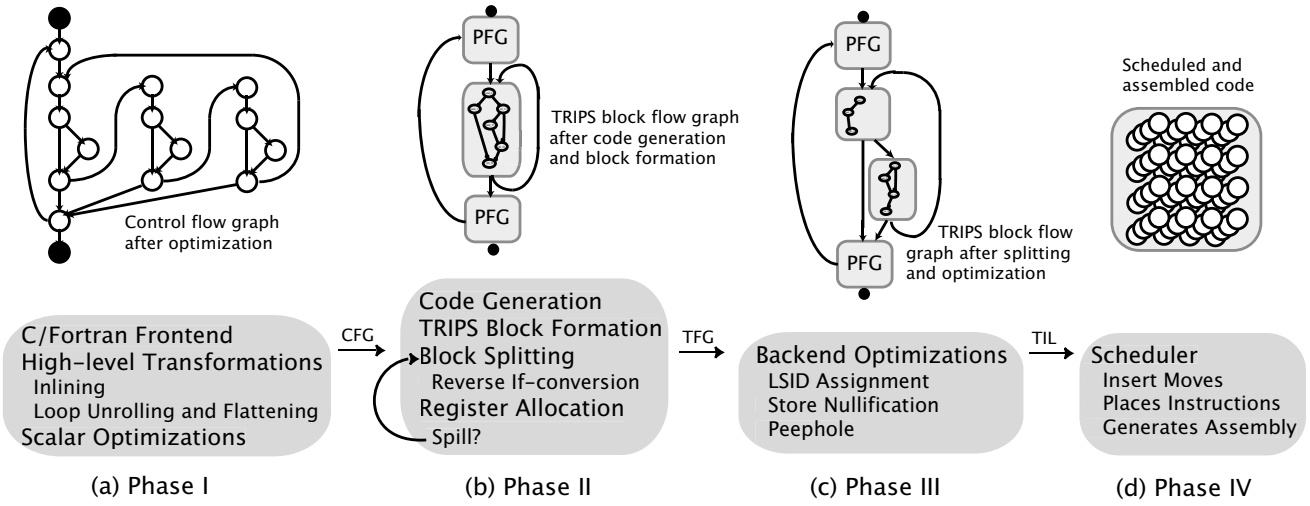

Figure 2. TRIPS Compiler Overview

Figure 3(b) shows the instructions in a TFG node before the PFGs are formed, and 3(c) shows the code in one PFG after block formation.

3.3. Block Formation

The architecture performs best if the compiler minimizes the number of TRIPS blocks and fills each one with useful instructions. The compiler thus tries to combine multiple TRIPS blocks into a single block. This step consists of two parts: (1) a policy for combining blocks (TFG nodes) to produce denser blocks, and (2) an analysis that, given the TIL instructions in a TFG node, estimates the number of instructions and LSIDs that will appear in the corresponding TASL. We call the latter step *size analysis*. Size analysis first estimates the number of instructions and LSIDs for each node in the TFG. The compiler then combines underutilized nodes, as long as the resulting block will not violate the instruction and LSID limits. Subsequent phases handle the register limit and constant output constraints.

Combining Blocks

To create full blocks, the compiler uses a greedy algorithm that merges parent and child TFG nodes unless:

1. The result would violate a block size or LSID constraint. To estimate whether merging two TFG nodes would violate these constraints, the compiler adds the estimates for the two nodes block size and LSID's together. This computation is an overestimate because subsequent optimizations may eliminate instructions.
2. The parent ends with a function call. Function calls must end inclusion down one control path to avoid jumping into the middle of a block.
3. The child has other TFG node predecessors. Without tail duplication, the compiler cannot merge children with multiple incoming edges.

Our current policy uses a top-down traversal of the TFG to combine each TFG node with as many of its children as will fit. The traversal begins at the root TFG node. When a TFG node is visited the compiler tries to merge it with its first child node. If it succeeds, it tries the next child, and so on. If all children can be merged, merging continues with the node's new children. When a merge fails the compiler moves on to the next TFG node in a breadth-first order. The resulting traversal is approximately breadth-first because merging changes the TFG as the traversal proceeds.

Figure 3(c) shows the block formation pass after combining the blocks in Figure 3(b), denoted by labels L0 and L1, into a single TRIPS block. The compiler deletes the branch to L1 reducing the block size. Notice that both remaining branches now target the same label and could be combined with instruction merging [24]. This example shows how block formation creates opportunities for further optimization.

Future enhancements to this algorithm include more discriminating merging policies. For example, if the block cannot accommodate all its children, a better policy chooses the subset that creates the most full block, or chooses the most frequently executed child. A policy that merges blocks on a hot path based on runtime profile information requires tail duplication to add blocks at control flow merge points, and is more likely to produce a set of blocks full of the most frequently executed instructions.

These policies are different than those for VLIW machines since the constituent instructions within the VLIW instruction must all be independent, and thus the goal of block formation is to create and combine independent instructions. The constituent instructions in a TRIPS block,

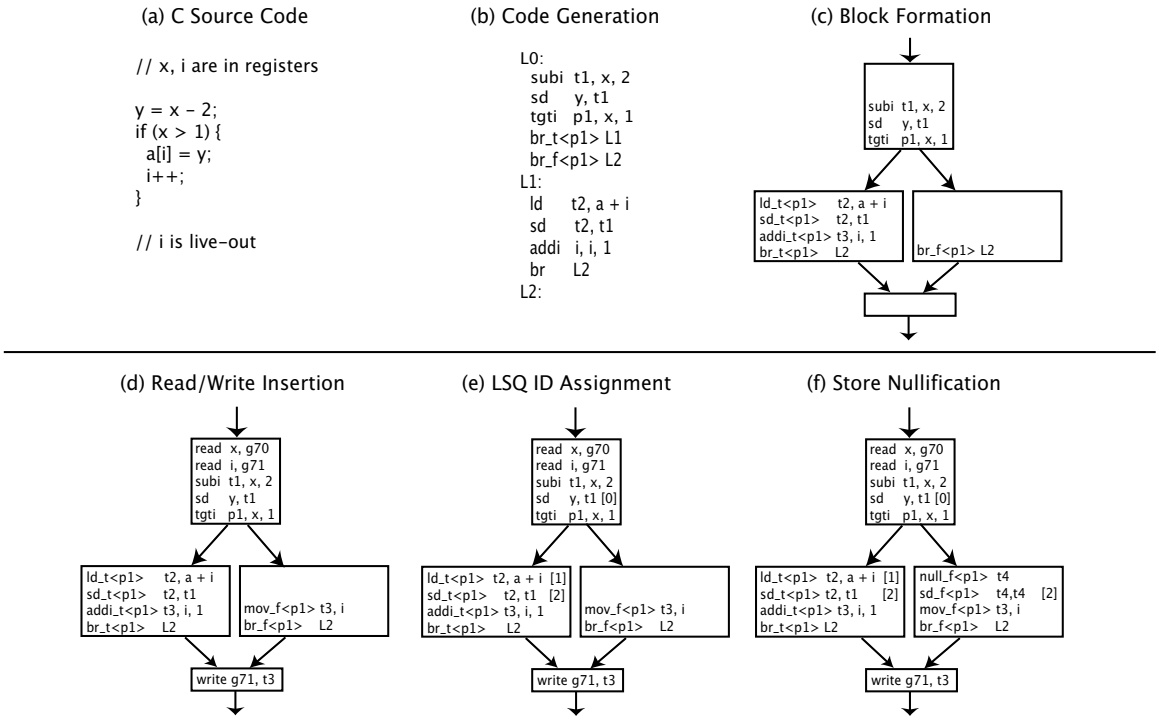

Figure 3. Transforming a Source Program Into a Legal TRIPS Block

conversely can be dependent, so the goal of block construction is to combine instructions that are likely to execute under the same conditions as well as creating independent instructions.

Size Analysis

Size analysis estimates the final number of TASL instructions from the TIL instructions and the number of LSIDs for a block. Two factors complicate this computation: (1) subsequent optimizations can add or remove instructions; and (2) TIL and TASL instructions do not have a one-to-one mapping.

When the compiler assigns LSIDs, it may need to insert additional instructions to guarantee the sequential memory semantics of a block. Size analysis assigns LSIDs and inserts any additional instructions before determining the number of instructions in a block. After the block size has been determined, the instructions inserted are removed from the block.

Translation inaccuracies derive from the difference in how TIL and TASL encode reuse. Consider a variable definition that five instructions subsequently use. In RISC ISAs and TIL, names express this reuse (e.g., a virtual register). In TASL's dataflow format, the defining instruction must explicitly name the location dependent operands. If an instruction has more consumers than target fields, the compiler must insert additional copy (*fanout*) instructions. To avoid inserting unnecessary copies, and since the ideal fanout tree topology is affected by the placement heuristics, the compiler postpones this step until it is scheduling the TASL.

TASL has three fanout instructions: `mov2`, `mov3`, and `mov4`, that can target two, three, or four instructions respectively, with correspondingly longer latencies. Which instruction the scheduler uses is based on the dataflow graph. Size analysis approximates fanout instructions assuming only `mov2` instructions. This estimate is not ideal, but anticipating the scheduler's decisions is challenging.

The first part of fanout estimation is straightforward. The compiler performs a depth-first search on the PFG to determine the number of uses for a particular register. Since the TRIPS blocks do not yet contain register read and write instructions, the compiler treats any live-in as a read instruction and any live-out as a write instruction. Next the compiler computes the fanout for an instruction by checking if the number of uses for the register defined by the instruction exceeds the number of targets for the instruction.

We use size analysis to drive block formation as described above, and in the block splitter.

3.4. Block Splitting

Three phases of the compiler may create TRIPS blocks that violate the block size or load/store constraints: (1) code generation; (2) spilling during register allocation; and (3) stack frame generation. If a TRIPS block violates either

constraint, the *block splitter* divides the block into multiple TRIPS blocks. The block splitter combines cutting–a method for splitting unpredicated regions of code, and reverse if-conversion [4]–a method for splitting predicated regions of code.

Block splitting begins with size analysis to determine if a block violates the block size or LSID constraint. Next, it traverses predicate blocks in the PFG in breadth-first order. It computes a running total, and continues to the next predicate block only if the current one does not violate the size and LSID constraints. The block splitter has two choices if a constraint is violated: it either cuts or reverse if-converts the predicate block. If the block is unpredicated (i.e., it is a basic block), then the block splitter cuts it by adding a branch and a label, creating two new TRIPS blocks that it adds to the TFG. If the block is predicated, then the block splitter must reverse if-convert [4] the predicate block from the TRIPS block. Reverse if-conversion creates a new TRIPS block beginning with the predicate block, removes the predicates from the instructions in the predicate block, and inserts a branch to it in the original TRIPS block. The compiler iterates until there are no violations.

After block splitting, all of the TFG nodes are guaranteed to meet the TRIPS block size and LSID constraints, but not the register bank constraint.

3.5. Register Allocation

Just like a RISC register allocator, an EDGE allocator assigns virtual registers to physical registers or spills to memory. However, unlike a conventional allocator, it need not assign those virtual registers whose live ranges are contained wholly within a block. This feature of the ISA reduces the number of register reads and writes.

The TRIPS register constraints pose a unique problem. There are 128 registers divided into four register banks with 32 registers apiece. Each TRIPS block is limited to eight register reads and eight writes per register bank (for a total of 32 register reads and 32 writes). The hardware register naming convention maps register names R0 to R127 to banks by interleaving them on the low-order bits of the register name. Compared to a graph coloring [8] or linear-scan allocator [16, 33], these features require additional state to track and enforce register bank constraints. Since the register banking constraint subsumes the fixed output constraint, we restrict our remaining discussion to satisfying the bank constraint. The compiler uses a modified linear-scan register allocator for TRIPS.

One way to pose this problem is to view each TRIPS block as a large instruction that can use and define 32 registers. We use this insight to compute live ranges on a block (TFG node) granularity. A virtual register is live if and only if it is defined and used in disparate blocks. The allocator gives priority to each virtual register based on its definitions, uses, and spill costs. Subsequently the allocator assigns virtual registers to physical registers or spills each live range in priority order.

The assignment phase uses an ordered list of available registers for each live range. We assign caller saved registers first by placing them at the front of the list. The allocator selects a register from this list and checks if it would cause any constituent TRIPS block to exceed its bank limitation. If not, it assigns the register to the live range. Otherwise, it excludes all registers from this bank from the list, and tries again until it finds an assignment or exhausts the list. If the allocator cannot satisfy the banking constraints in all blocks, it spills.

Spilling introduces load and store instructions, and thus may cause a TRIPS block to violate the block size or load/store constraints. The block splitter therefore reexamines every block in which the allocator inserted spill code. The compiler splits any block with a violation and then repeats register allocation. Iterative block splitting is guaranteed to terminate eventually since it strictly reduces the size of a block. The current allocator rarely spills due to bank constraints. However, more aggressive TRIPS block formation may expose the need for additional enhancements or a graph coloring allocator that tends to spill less [33].

3.6. Memory Ordering and Block Termination

To guarantee sequential memory semantics, phase III of the compiler assigns a unique ordering identifier (LSID) to each load and store in a TRIPS block. The microarchitecture ensures that operations to the same address commit in order [30]. The compiler can satisfy this constraint by simply assigning an identifier to every load and store in breadth-first program order. However, the compiler is free to reuse LSIDs if it can guarantee that only one load or store will ever fire for each LSID. If the compiler reuses a LSID, it can include more loads and stores in a block, increasing the block size and reducing register pressure. The only restriction is that loads and stores cannot share an LSID because the block termination logic uses store LSIDs to detect completion.

The compiler's LSID assignment algorithm modifies a breadth-first program order pass by dividing the graph into levels. For each level it maintains n ordered lists of loads and stores, where only one of the n lists will execute (e.g., for an if-then-else, it creates a level with one list for the true branch and one for the false). Without loss of generality, assume the first list starts with a load. The algorithm assigns this first load a LSID. It then checks the top of the other lists for loads and assigns them the same LSID. It then assigns the next LSID and so on, until it exhausts the first list. It then exhausts the second list, and so on. When it finishes all the loads and stores at the current level, it moves to the next level.

	O3			O4			Hand		
	min	max	mean	min	max	mean	min	max	mean
SPEC2000	4.9	42.4	12.6	8.4	46.7	18.5	-	-	-
EEMBC	5.0	32.6	9.3	8.2	49.9	22.7	-	-	-
Microbenchmarks	3.7	101.3	31.6	16.5	101.3	48.6	17.03	112.5	73.5

Table 1. Dynamic Instructions/Block

The compiler must also guarantee that all paths produce the same store LSIDs and same register writes to satisfy the fixed output constraint: once the block produces all its outputs, it completes. For every store LSID in the TRIPS block, the compiler simply enumerates all paths and checks whether it contains a store with the same LSID. If not, it inserts a *null store* for the missing LSID. The TRIPS ISA defines a null instruction that signals to the hardware no value will be produced for a store LSID or a register write. A null store is a store instruction that receives as input a null instruction.

The compiler solves this same problem for register writes by converting into SSA form. As it goes into SSA form, the compiler identifies any upwardly exposed register uses in a block and inserts a corresponding register read. Then the compiler inserts copies on all the paths without definitions. Figure 3(d) shows the TRIPS block after the allocator inserts read and write instructions and after SSA renaming. Notice the `mov` instruction which is in the predicate block on the false path through the TRIPS block. When SSA eliminates ϕ nodes, it produces exactly the semantics the compiler needs to ensure that all paths produce the same set of outputs.

Figure 3(e) shows how the compiler assigns LSIDs. There are three LSIDs: 0, 1 and 2 to a store, load and another store. The first store is unpredicated and will produce a value down all paths for LSID 0; the second store however is predicated on p1 being true but there is not a corresponding LSID down the other path through the TRIPS block. In Figure 3(f), the compiler inserts a null store on the false path with LSID 2 so that the same store mask is produced down both paths. At this point, the TRIPS block is legal and is ready to be scheduled.

3.7. Placement and Assembly

In phase IV, the scheduler maps TIL instructions to execution tiles, and then encodes its placement decisions in TASL [34]. Since, for a given implementation, the instruction number assignment determines where on the ALU array the instruction will be placed, the compiler can implicitly reason about the interconnection topology and delay. The instruction scheduler thus seeks to minimize inter-instruction latency and contention for an issue slot on the same ALU. Previous work presents a greedy scheduling algorithm for these constraints [27]. After scheduling, the assembler and linker create an ELF binary.

4. Results

We use two simulators to produce experimental results. The first is a fast ISA functional simulator, used to generate average dynamic block sizes and demonstrate that the compiled code is correct. For these experiments, we use all 30 of the EEMBC 2.0 benchmarks and 21 of the 26 SPEC2000 benchmarks (the five remaining benchmarks are FORTRAN90 and C++ which Scale does not support). For timing experiments, we use a near-cycle-accurate simulator, called *tsim-proc*, that closely models the TRIPS hardware (a recent performance evaluation estimated that tsim-proc and the TRIPS RTL differ by under 4% on average). Tsim-proc faithfully models most delays in the TRIPS prototype implementation, including a one-cycle hop from any tile to an adjacent tile, a 32KB, 2-way set-associative L1 data cache with a 2-cycle latency, a 64KB, 2-way set-associative L1 instruction cache with a 1-cycle latency, an 8-cycle block fetch latency, and a 3-cycle branch prediction latency.

Since tsim-proc is slow compared to higher-level performance simulators, we perform all timing analysis on a set of microbenchmarks. They consist of 15 frequently executed loops extracted from SPEC2000 programs, four kernels from a MIT Lincoln Labs radar benchmark (doppler_GMTI,ff2_GMTI, fft4_GMTI, transpose_GMTI), a vector add kernel (vadd), a ten by ten matrix multiply (matrix_1), and a prime number generator (sieve). All code is compiled with full optimizations (O3), and a second set of compilations (O4), extend the O3 level with predicated TRIPS block formation as described in Section 3.3. Finally we show results for hand-assembled versions of the microbenchmarks.

4.1. Correctness

The instruction-level simulator produces a block profile that captures the number of blocks and the dynamic number of instructions that execute, i.e., instructions that do not have false predicates. Table 1 shows the dynamic averages, maximum, and minimum number of instructions in each block for optimization levels O3 and O4, and also hand-assembly levels for the microbenchmarks. These results show that the O4 results provide a significant improvement in TRIPS block size over O3, but that there remains a large gap for the microbenchmarks between the O4 and hand-optimized results.

Figure 4 shows the dynamic block sizes on a per-benchmark basis. Many of the loops show significant increases when the compiler combines TRIPS blocks at O4. Those that still show remaining gaps between the 04 level and hand assembly suffer from suboptimal unrolling of for loops and the absence of while loop unrolling; ammp_1, ammp_2, art_1, art_2, and gzip_2 are notable examples.

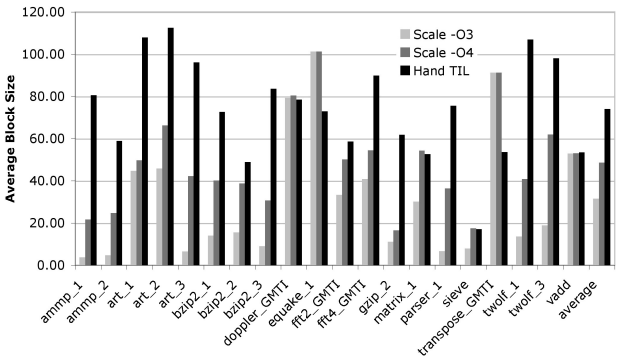

Figure 4. Average Dynamic Block Size

4.2. Performance Analysis

As a performance comparison baseline, we measured the microbenchmarks compiled with gcc at full optimization (-O3), on a DS-10L workstation with an Alpha 21264 processor [21]. We compare microbenchmark cycle counts of the two systems, selecting the Alpha as a comparison point because it executes programs in an efficient number of cycles, but is also an aggressively clocked implementation (for its process technology). The only parameters in tsim-proc that differ from the TRIPS implementation are in the memory subsystem, which we set to approximate the equivalent structures in the DS-10L workstation and make a fairer comparison. The parameters we used were a 2MB, 2-way set-associative L2 cache with a 12-cycle hit latency and a 60-cycle main memory latency.

Figure 5 shows TRIPS speedups over Alpha at the O3, O4, and hand-assembled levels. Comparing the trends between Figures 4 and 5 reveals that increases in dynamic average block size correlate well with improved performance. Only transpose_GMTI counters this correlation. The average results show that with basic blocks as TRIPS blocks (O3), the microbenchmarks run 42% faster than Alpha (comparing cycle counts directly). With multiple basic blocks in a single TRIPS blocks (O4), the microbenchmarks run 41% faster than 03, and 83% faster than on Alpha. A large performance gap exists, however, between the hand-coded microbenchmarks and the compiler; they run a full 3.2 times faster than the Alpha. The hand-coded microbenchmarks show an average TRIPS IPC of 3.2, with a maximum IPC of 4.6; the number of TRIPS to Alpha instructions are comparable.

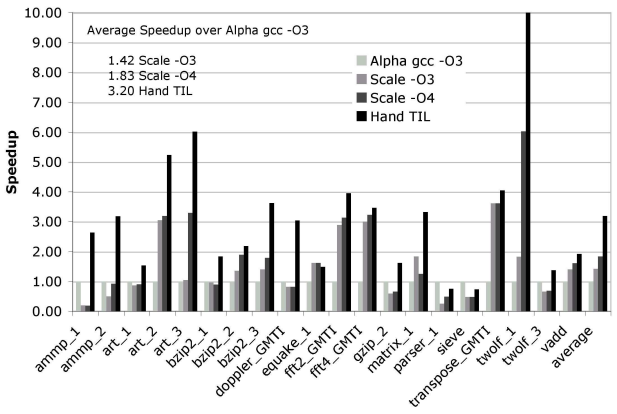

Figure 5. Performance Comparison

A few individual results stand out. On two benchmarks (parser_1 and sieve) all versions perform worse than Alpha. These benchmarks are highly serial, offering little ILP. There is one long dependence chain in parser_1 that the O4 compiler can capture in a block with 36 instructions (in the hand-coded version the block grows to 76 instructions), but the lack of sufficient ILP exposes the TRIPS block mapping overhead. Even the hand-coded version of sieve has fewer than 20 instructions in the block on average, again exposing the dynamic block overheads. One benchmark, matrix_1, shows a slowdown when going from O3 to O4 compiled code. There is a single if-converted test inside a loop that is never true. So the inclusion of the test in the loop body only adds contention for resources. The compiler could determine with profiling that the test is not on the critical path and choose not to include it in the loop. The performance would then be the same for O3 and O4 compiled code.

Five hand-coded microbenchmarks (art_2, art_3, ff2_GMTI, transpose_GMTI, and twolf_1) perform between a factor of four and ten better than Alpha. Six O4 compiled microbenchmarks (art_2, art_3, ff2_GMTI, ff4_GMTI, transpose_GMTI, and twolf_1) improve by a factor of at least three over Alpha. An interesting case is ff2_GMTI; although the blocks are not full (59 instructions for hand-coded, and 48 for O4), the processor successfully employs speculation to exploit loop-level parallelism across blocks, in addition to unrolled loop-level parallelism within blocks and ILP within individual loop bodies.

5 Related Work

The work on TRIPS blocks builds on previous work on compiling predicated hyperblocks for VLIW machines [24, 23, 29]. VLIW architectures build hyperblocks to maximize exposure of independent instructions for long-word packing. When forming hyperblocks, VLIW compilers scrutinize dependence height in less-frequently-accessed basic blocks, since that height puts a lower bound on the VLIW

schedule. TRIPS blocks differ in two subtle ways: first, the four block restrictions limit the hyperblocks that can legally be formed; second, while both classes of architectures want hyperblocks to be full of many useful, independent instructions, dependence height is a non-issue for TRIPS blocks, since blocks can be committed and deallocated as soon as all of their outputs are received, regardless of what is happening down other predicate paths in the block.

EDGE architectures are a hybrid of dataflow and sequential machines, using dataflow within a block, conventional register semantics across blocks, and conventional memory semantics throughout. An EDGE compiler thus differs significantly from compilers for pure dataflow machines [3, 18], since dataflow machines limited the programming model to a functional one where programs cannot produce multiple values in the same location, thus relieving the compiler and architecture of the memory disambiguation problem.

Recent work on compilers for dataflow-like architectures is similar to TRIPS [6, 7, 11]. The most closely related is the CASH compiler that targets a substrate called ASH, for application-specific hardware. Like the TRIPS compiler, CASH's Pegasus intermediate representation targets a predicated hyperblock form translated into an internal SSA representation, compiling small C and FORTRAN programs. Many of the instruction-level transformations, using Pegasus' GSA-like decoded multiplexors, are applicable to TRIPS blocks. Two major differences between the TRIPS and CASH compilers are the hardware targets and the block restrictions. The CASH compiler targets a hardware synthesis tool flow, whereas the TRIPS compiler targets a specified ISA running on a fixed microarchitecture. Therefore, the CASH compiler can produce mostly unconstrained blocks, except for chip area constraints. The difference between unbounded graphs for a co-designed substrate (Pegasus/ASH) versus limited graphs for a fixed substrate (EDGE ISAs) dramatically changes the compilation problem.

The WaveScalar architecture [32] forms "waves" that are similar to hyperblocks except for the mechanism that executes subsequent graphs (polypath wave execution rather than a single speculative flow of control). A second, more minor difference is the architectural mechanism used to enforce sequential memory semantics (instruction pointers in WaveScalar as opposed to load/store sequence numbers in TRIPS). The third major difference with the TRIPS architecture is that WaveScalar publications advocate dynamic, run-time placement of instructions [31, 32], as opposed to the static TRIPS approach of mapping compiler-assigned instruction numbers to specific ALUs, thus permitting the compiler to optimize for operand routing distance [27].

The RAW architecture [36] is a tiled architecture whose focus on technology-scaling motivations is similar to TRIPS. Both the RAW and TRIPS compilers focus on instruction locality, placing dependent instructions on the same, or nearby, execution tiles. However, the two architectures' execution models are fundamentally different. RAW targets fine-grained parallelism by compiling programs into many threads of control, running on multiple tiles under a strict static schedule. TRIPS instead uses a single program counter, maps instructions from the same TRIPS block to multiple tiles, and dynamically executes instructions based only on their dataflow constraints.

6. Conclusions

The block-atomic execution model used by EDGE architectures provides potential performance and energy advantages compared to traditional ISAs, but also presents new compilation challenges. In particular, the compiler must generate full blocks of useful instructions but still obey the block constraints imposed by the ISA. In this paper, we described the compiler flow implemented to generate code that adheres to the TRIPS block constraints.

With this flow and a set of algorithms used to adhere to the block requirements, Scale was able to compile and run all of the SPEC2000 C and FORTRAN77 and EEMBC benchmarks, and to grow the blocks substantively over basic blocks. On a set of microbenchmarks, the compiler was able to outperform an Alpha 21264 running fully optimized gcc code by 83%.

While some of the microbenchmarks demonstrated large blocks, others showed block sizes that were still small. A comparison with hand-assembled benchmarks shows that several more features must be added to the compiler to approach the 3.2-fold speedup over the 21264 produced by the hand-assembled benchmarks, in particular while-loop unrolling, while-loop peeling, and predicated instruction merging.

References

[1] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, Jan. 1983.

[3] K. Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[4] D. I. August. *Systematic Compilation for Predicated Execution*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Feb. 2000.

[5] D. I. August, D. A. Connors, S. A. Mahike, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th*

International Symposium on Computer Architecture, pages 227–237, Barcelona, Spain, June 1998.

[6] M. Bidiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical report, CMU, May 2002.

[7] M. Bidiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *Proceedings of the 2004 Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[8] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Dept. of Computer Science, Rice University, Apr. 1992.

[9] M. Budiu. *Spatial Computation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2003.

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

[11] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, pages 62–69, Apr. 2000.

[12] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. Rau, and M.Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *Proceedings of the 29th International Symposium on Microarchitecture*, Paris, France, Dec. 1996.

[13] W. Y. Chen, S. A. Mahlke, N. J. Warter, S. Anik, and W. W. Hwu. Profile assisted instruction scheduling. *International Journal of Parallel Programming*, 22(2):151–181, April 1994.

[14] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, Washington, DC, USA, 1998. IEEE Computer Society.

[15] P. Craig, R. Crowell, M. Liu, B. Noyce, and J. Pieper. The Gem loop transformer. *Digital Technical Journal*, 1999.

[16] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, Jan. 1989.

[17] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the Second International Symposium on Computer Architecture*, pages 126–132, New York, NY, USA, 1975.

[18] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[19] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide-issue processors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 266–276, January 1998.

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

[21] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct. 2002.

[22] S. A. Mahlke. *Exploiting instruction level parallelism in the presence of branches*. PhD thesis, UIUCCS, 1996.

[23] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Portland, OR, Dec. 1992.

[24] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture*, pages 181–193, June 1997.

[25] R. Nagarajan, D. Burger, K. S. McKinley, C. Lin, S. W. Keckler, and S. K. Kushwaha. Instruction scheduling for emerging communication-exposed architectures. In *The International Conference on Parallel Architectures and Compilation Techniques*, pages 74–84, Antibes Juan-les-Pins, France, Oct. 2004.

[26] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 40–53, Dec. 2001.

[27] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 257–271, New York, NY, USA, 1990.

[28] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 422–433, June 2003.

[29] S. Swanson, K. Michaelson, and M. Oskin. Configuration by combustion: Online simulated annealing for dynamic hardware configuration. In *ASPLOS Wild and Crazy III*, October 2002.

[30] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the 36th Symposium on Microarchitecture*, December 2003.

[31] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 142–151, Montreal, June 1998.

[32] TRIPS Team. TRIPS assembler and assembly language (TASL) specification. Technical report, University of Texas, 2005. http://www-cs.utexas.edu/users/cart/trips/spec/.

[33] TRIPS Team. TRIPS intermediate language (TIL) manual. Technical report, University of Texas, 2005. http://www--cs.utexas.edu/users/cart/trips/spec/.

[34] 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.

[35] Z. Wang, D. Burger, K. S. McKinley, S. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 388–398, San Diego, CA, June 2003.