The Dissertation Committee for Aaron Lee Smith
certifies that this is the approved version of the following dissertation:

Explicit Data Graph Compilation

Committee:

Douglas C. Burger, Supervisor

Lizy John

Stephen W. Keckler

Calvin Lin

Kathryn S. McKinley

Explicit Data Graph Compilation

by

Aaron Lee Smith, B.A., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2009

Acknowledgments

This research would not have been possible without the support of many people. First, I would like to thank my advisor Doug Burger for his advice and friendship throughout my time in graduate school. I would like to thank my remaining committee members for their service and mentorship–Steve Keckler, Calvin Lin, Kathryn McKinley, and Lizy John. A special thanks to Calvin Lin for advising me during my first few years in graduate school, and to Steve Keckler and Kathryn McKinley for their close collaboration on the TRIPS project over all these years.

I thank the entire TRIPS team for their collaboration–Robert McDonald, Jim Burrill, Karu Sankaralingam, Ramdas Nagarajan, Bill Yoder, Nitya Ranganathan, Raj Desikan, Saurabh Drolia, Madhu S. Sibi Govindan, Divya Gulati, Paul Gratz, Heather Hanson, Changkyu Kim, Haiming Liu, Premkishore Shivakumar, Mark Gebhart, Katie Coons, Bert Maher, Madhavi Krishnan, Sundeep Kushwaha, Behnam Robatmili, Sadia Sharif, and Simha Sethumadhavan. What we have accomplished together has been truly amazing. A special thanks also to all the contributors to the Scale compiler which this research is based upon.

I thank the FASTRAC team for giving me another home in WRW across the street from ACES–Jamin Greenbaum, Sebastian Munoz, Thomas

Campbell, Greg Holt, Shaun Stewart, Tena Wang, and Glenn Lightsey.

I thank my colleagues at Microsoft Research–Gabriel Loh, Andrew Putnam, and Karin Strauss–for their thoughtful feedback and patience while I completed this work. I especially thank Weidong Cui for helping me scramble at the last minute to fill out departmental forms. As well as Don Porter for couriering them personally back to Austin.

I think all my friends for their support, in particular–Robert Anderson, Paul Navratil, Benjamin Lee, Dan Muldoon, Hiroshi Sasaki, Vijay Janapa Reddi, Masaaki Kondo, and Judson Hamilton.

Finally, I thank my family for their encouragement.

Aaron Lee Smith

The University of Texas at Austin
December 2009

Explicit Data Graph Compilation

Publication No. _____

Aaron Lee Smith, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Douglas C. Burger

Technology trends such as growing wire delays, power consumption limits, and diminishing clock rate improvements, present conventional instruction set architectures such as RISC, CISC, and VLIW with difficult challenges. 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 concurrency.

In this research we develop the first compiler for an Explicit Data Graph Execution (EDGE) architecture and show how to solve the new challenge of compiling to a block-based architecture. In EDGE architectures, the compiler is responsible for partitioning the program into a sequence of structured blocks that logically execute atomically. The EDGE ISA defines the structure of, and the restrictions on, these blocks. The TRIPS prototype processor is an EDGE architecture that employs 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).

The challenges addressed in this thesis are twofold. First, we develop the algorithms and internal representations necessary to support the new structural constraints imposed by the block-based EDGE execution model. This first step provides correct execution and demonstrates the feasibility of EDGE compilers. Next, we show how to optimize blocks using a dataflow predication model and provide results showing how the compiler is meeting this challenge on the SPEC2000 benchmarks. Using basic blocks as the baseline performance, we show that optimizations utilizing the dataflow predication model achieve up to 64% speedup on SPEC2000 with an average speedup of 31%.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Technology trends such as growing wire delays, power consumption limits, and diminishing clock rate improvements, present conventional 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 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 [29, 45].

Explicit Data Graph Execution (or EDGE) architectures partition the work between the compiler and the hardware differently than RISC, CISC, or VLIW architectures [1, 16, 36, 53, 62–65, 73], 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 [47]. The EDGE ISA defines the structure of, and the restrictions on, these blocks. Forming structurally correct blocks is sufficient for execution, however blocks must also be optimized to achieve high performance. Second, instructions within each block employ *direct instruction communication*. The compiler encodes instruction dependences explicitly, eliminating the need for the hardware to discover most dependences dynamically. Previous work describes the research on direct instruction communication [20, 52]. This work focuses on the former, the compiler flow and algorithms necessary to generate correct [68] and optimized blocks [42, 71].

The structural restrictions on blocks permit simpler hardware, but are more restrictive than those for traditional hyperblocks [45] and superblocks [27]. These restrictions make the compiler's task of forming dense but legal blocks more challenging. Fewer restrictions allow for a simpler compiler but require

more complicated hardware. The TRIPS processor is the first EDGE architecture. The TRIPS ISA employs 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).

The TRIPS hardware issues instructions dynamically and out-of-order as their source operands become available. In addition, the ISA employs a lightweight *dataflow predication* [71] model, necessary for forming large blocks. Predication linearizes instruction flows by converting control dependences to data dependences, thus improving control flow predictability, instruction fetch bandwidth, and the size of the instruction scheduling window for the compiler. VLIW and vector machines have successfully applied predication to obtain all three of these improvements [24, 61, 79]. However, predicated execution has not achieved widespread use in out-of-order architectures. The complexities of merging predication with dynamic scheduling [56]–particularly register renaming [18, 35, 40, 78]–have outweighed its perceived benefits. Block atomic execution employing dataflow predication is one solution to these problems. In dataflow predication, any instruction producing a value can instead produce a predicate. With this ISA support, as well as appropriate support in the microarchitecture and compiler, the TRIPS processor exploits the benefits

afforded by both predication and dynamic out-of-order issue.

1.1 TRIPS EDGE ISA and Microarchitecture

This section provides an overview of the TRIPS EDGE ISA and microarchitecture [16, 36, 52, 53, 63]. 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. For example, in a RISC architecture, an ADD instruction adds the values of R4 and R5 and places the result in R3:

```
ADD R3, R4, R5
```

An equivalent 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 (both within the same block). Upon receiving its operands and predicate, instruction 37 will fire only if the predicate condition evaluates to true. We call the latter representation *target form*.

To provide some context for the compiler's compilation target, Figure 1.1 shows the TRIPS microarchitecture. The execution core consists of an

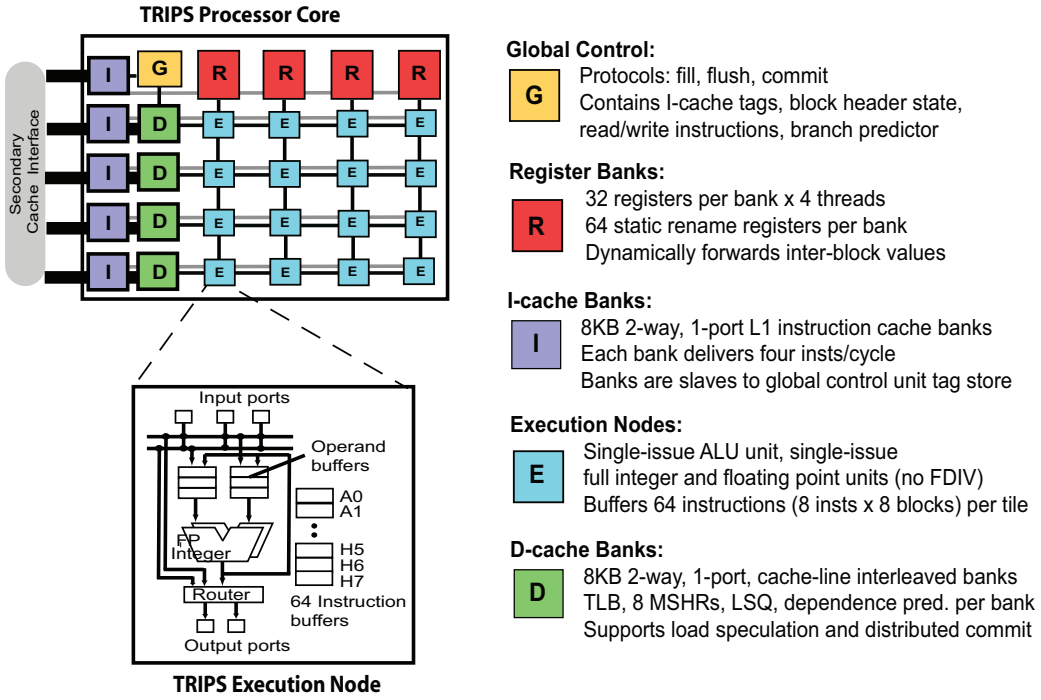


Figure 1.1: TRIPS Prototype Processor Core

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

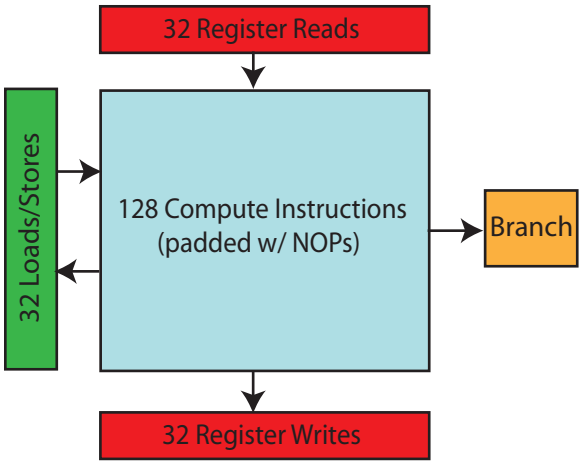


Figure 1.2: TRIPS Block Constraints.

1.2 What is a Block?

The TRIPS microarchitecture supports out-of-order execution of instructions organized within blocks. The compiler is responsible for forming blocks that adhere to the following architectural constraints (Figure 1.2):

- **Fixed Block Size:** All blocks contain at most 128 compute instructions (register reads and writes are additional). Blocks that do not contain 128 compute instructions are padded with NOP's.
- **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 and store instructions per block, since loads or

stores down disjoint predicate paths may share the same LSID, but at most one memory operation with a given LSID may fire, and there are at most 32 LSIDs. If an LSID is shared, it may only be shared among like memory operations. In other words, a load and store cannot share the same LSID.

- **Register Constraints:** There are 128 registers divided into four register banks each with 32 registers. Each register bank issues at most eight read and eight write instructions per block. This means that each block can read from 32 global registers and write to 32 global registers. All register accesses are done through `read` and `write` instructions which do not count toward the 128 instruction limit.

- **Constant Output:** In order for the control logic to detect that a block is complete, each block emits a consistent number of register writes and stores, plus exactly one branch. If a block contains a store or register write down one path of execution but not another, the compiler must insert additional instructions on the alternative path to ensure the hardware does not wait for a store or write that will never issue. This restriction adds instruction overhead, but considerably simplifies detection of block termination.

Figure 1.3(a) shows two disjoint paths of execution. Although there are three memory instructions in the example, the compiler can take advantage of the fact that only one of the paths will execute and assign the same IDs

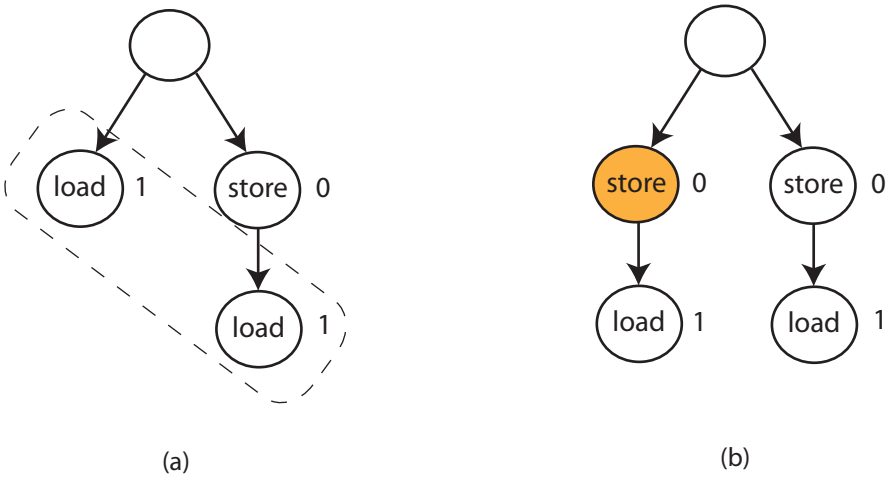


Figure 1.3: Example of (a) loads down disjoint predicate paths sharing the same LSID, and (b) store inserted by the compiler for block completion.

to both loads. This example shows how the compiler can reduce the number of LSIDs required for a block to include more than 32 static load and store instructions. Figure 1.3(b) shows the same example but with an additional store instruction inserted by the compiler. Because each path of execution in a block must produce the same set of store IDs, the compiler must add an additional store with LSID 0 to the block. The compiler must perform a similar operation for register writes.

1.3 Dataflow Predication Model

In order to increase the size of blocks to improve performance, TRIPS uses a dataflow predication model. In this model any instruction that pro-

duces a value can instead produce a predicate. The operand network routes predicates to their dependent instructions just like any other value. Dataflow predication makes possible a clean synergy between predication and out-of-order execution, with lower predicate overhead than the partial predication implemented in previous dataflow architectures [4, 10, 25, 37, 74], and lower hardware complexity than proposed predicated out-of-order superscalar designs [18, 44, 56, 78].

The compiler for the TRIPS ISA must follow a number of rules to produce well-formed, predicated blocks:

1. Any instruction (except for a few specific data movement and constant generation instructions) may be predicated. A two-bit predicate field indicates whether an instruction is predicated and on what polarity of the arriving predicate the instruction should be executed.

2. For a predicated instruction to fire and execute, it must receive all of its data operands and a matching predicate operand. A matching predicate is one that matches the polarity of the waiting instruction. For example, an instruction waiting for a "false" predicate will only fire when a "false" predicate arrives.

3. Multiple instructions may target the predicate operand of an instruction, but at most one may deliver a matching predicate.[1]

[1]Multiple instructions may target the same operand field of any instruction, as long as the compiler guarantees only one value is ever received for an operand.

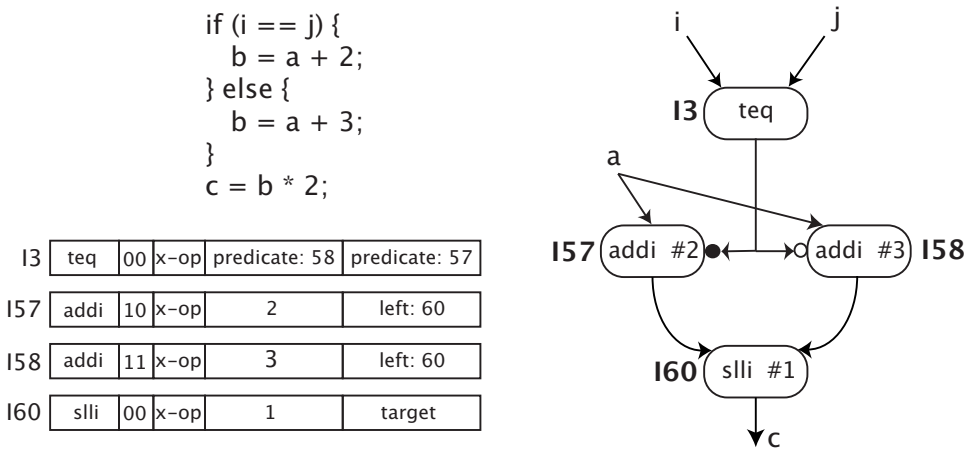


Figure 1.4: Predication in the TRIPS ISA.

4. The predicated dataflow graphs must preserve the exception behavior of an unpredicated program, meaning that the same exceptions must be detected at the TRIPS block boundaries.

This thesis treats all predicates as *intra-block* values. There are subtle architectural restrictions on using predicates across block boundaries that are not discussed here. Throughout the remainder of this document, we use the following syntax for predication. To indicate that an instruction is waiting for a "false" predicate, we append "`_f`" to the opcode name. Likewise, we append "`_t`" to indicate a "true" predicate. The predicate an instruction is predicated on is shown in "`<>`". For example,

```
addi_t<p100> t2, t17, #1
```

is an add immediate instruction that is predicated on `p100` being true. If the predicate matches then one is added to the value in `t17` and stored in `t2`. Note that `p100`, `t2` and `t17` are not registers but dataflow edges in the data dependence graph. For convenience, we distinguish between operands that represent temporary values with a "`t`" and those that represent predicates with a "`p`".

1.4 Compiling to TRIPS

To compile to the TRIPS ISA, the compiler forms blocks of instructions that adhere to the block constraints. The compiler forms hyperblocks, using the dataflow predication model, to expose parallelism and take advantage of the large number of compute resources available. The example in Figure 1.5 shows the compilation of a C function to the TRIPS ISA. A RISC architecture uses registers as temporary storage for values between computations. In an EDGE ISA, registers are only used to communicate values *between* blocks of instructions. Computation within a block is in dataflow fashion with an instruction sending its result directly to the instructions which use that result.

In the TRIPS compiler, the back end first generates basic blocks in a RISC-like intermediate form (Figure 1.5(b)), and the basic blocks are then combined into hyperblocks. Figure 1.5(c) shows the corresponding dataflow graph for the function after hyperblock formation. The two paths of execution in the original source have been combined into a single predicated region of code. The `tgti` instruction creates a predicate that is used to predicate both

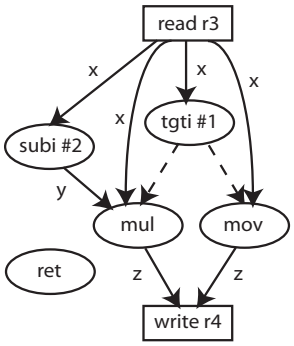
```
int rei(int x) {
  int z;
  int y = x - 2;
  if (x > 1) {
    z = x * y;
  } else {
    z = x;
  }
  return z;
}
```

(a) Example C code

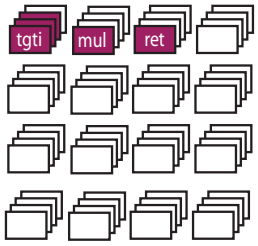
```
// r2 = y
// r3 = x
// r4 = z

L0: subi r2,r3,#2
    ble r3,#1,L1
    mul r4,r3,r2
    br L2
L1: mov r4,r3
L2: ret
```

(b) RISC assembly



(c) Dataflow graph



(d) Scheduling

```
R0: read I0,I1,I2 [R3]
I0: subi I3
I1: tgti I2,I3
I2: mov W0
I3: mul W0
I4: ret
W0: write [R4]
```

(e) TRIPS assembly

Figure 1.5: Example of compiling a C function to the TRIPS ISA.

the `mul` and `mov` instructions. If the test is true (i.e., `x > 1`) then the multiply instruction will execute. Otherwise the `mov` instruction will execute. Of special interest is the return instruction. Since this instruction is not dependent on any other instruction, the `ret` will execute immediately, but the block will not commit until the register write is also complete.

After legal blocks have been formed that adhere to the TRIPS ISA, the scheduler determines the execution tile and reservation station for each instruction. In the TRIPS ISA, instructions are statically scheduled but execute dynamically, out-of-order. The scheduler converts the instructions from a more traditional RISC-like intermediate form into the TRIPS target form called TASL, which encodes the consumers of each instruction (Figure 1.5(e)).

1.5 Thesis Statement

Before this research was undertaken, it was unknown whether an EDGE compiler could be built that would generate code with acceptable code quality, compile time, and compiler complexity. This dissertation is the first to solve the problem of compiling to a block-based EDGE architecture and contributes the overall compiler flow, the internal representation for predicated blocks, and the algorithms for forming legal blocks and optimizing them. Specifically, this dissertation describes the high-level back-end flow of the TRIPS compiler, evaluates the resultant code quality, and evaluates some of the design decisions made in the TRIPS ISA

1.6 Dissertation Contributions

I led the development of the compiler support for the TRIPS ISA, but the design and implementation of any compiler is a large task, and many people have contributed to the TRIPS compiler project. In this section, I highlight my specific contributions in the context of related work.

The basis of the TRIPS compiler is the Scale compiler developed originally at the University of Massachusetts Amherst and now at the University of Texas at Austin. I joined the TRIPS compiler project at the onset in the summer of 2002, and during that time, ported Scale's Alpha backend to a rudimentary version of the TRIPS ISA, while working with Steve Keckler, Doug Burger, Kathryn McKinley, Jim Burrill, Bill Yoder, Robert McDonald, Chuck Moore, Ramadass Nagarajan, and Karthikeyan Sankaralingam to define the ISA for the TRIPS prototype processor. I wrote the TRIPS Application Binary Interface (ABI), which Jim Burrill extended, and led the development of the TRIPS Intermediate Language (TIL). I ported the diet libc [77] C library to TRIPS and wrote the runtime system. Bill Yoder, Mark Gebhart, and I ported the math library and support for software floating point.

Jim Burrill, Jon Gibson, and I updated the code generator to use the opcodes for the TRIPS ISA. I developed all the support in the backend for predicated blocks including the internal representation used, the algorithms for entering block form, the routines for analyzing block constraints, and the block splitter which reforms illegal blocks into legal ones. Only an overview of this research has been previously published [68].

Jim Burrill wrote the original linear scan register allocator used by all the back ends. I modified the register allocator to work on the compiler's intermediate representation for predicated blocks, wrote the dataflow analysis used to compute liveness on block form, added support to the register allocator for the blocks constraints, and wrote the support for handling blocks that become illegal due to spill code. I also recognized that the register allocator could minimize the number of blocks split due to spilling by prioritizing which live ranges to assign based on the block constraints. Behnam Robatmili and I collaborated to implement a priority function that assigns live ranges based on block size which is published in [11]. In Chapter 4, I present a new, unpublished priority function, that combines live range length with the block constraints.

I wrote the backend support for predication and the optimizations utilizing the dataflow predication model [71], and implemented the hyperblock generation framework that incrementally if-converts, combines, and optimizes blocks. Bert Maher extended the hyperblock generator to support a generalized form of loop unrolling and to explore policies for selecting which blocks to merge [42].

The software scheduler was developed by Katherine Coons, Ramadass Nagarajan, Xia Chen, Sundeep Kushwa, and Saurabh Drolia. Bill Yoder ported the GNU assembler and linker to TRIPS [83]. Innumerable students, staff, and users helped to find and fix bugs.

1.7 Dissertation Layout

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of the TRIPS compiler. Chapters 3 and 4 describe how to generate correct code, and Chapter 5 describes how to generate optimized code. Finally, Chapter 6 evaluates the complexity of the algorithms developed in this research, the resultant generated code quality, and some of the design decisions made in the TRIPS ISA.

Correct Code: Chapter 3 explains how the compiler represents and reasons about blocks of predicated code and how it forms blocks based on the structural constraints imposed by the block-based EDGE execution model. To simplify the discussion, this chapter assumes the TRIPS architecture supports an unlimited number of registers, load-store identifiers, and instructions per block. The chapter develops the algorithms for computing liveness on block form, inserting read and write instructions, assigning load-store identifiers, nullifying write and store instructions, and also shows how the static single assignment form (SSA) of a predicated block can be built.

Next, Chapter 4 describes how the compiler handles blocks that violate any of the TRIPS block constraints. This chapter introduces *block splitting*–a framework for reshaping blocks that violate the block size and load-store identifier constraints. Block splitting is an adaption of reverse if-conversion, which was developed for predicated VLIW architectures. The chapter explains how to perform register allocation on blocks and handle spilling, which may cause a block to violate a constraint. Finally, stack frame generation is described.

Together, Chapters 3 and 4 provide all the necessary details for building a working EDGE compiler for the TRIPS ISA.

Optimized Code: Chapter 5 turns to building optimized predicated blocks and presents *iterative hyperblock formation*–a solution for forming large blocks with respect to the additional constraints imposed by EDGE architectures. Iterative hyperblock formation solves the phase ordering problem which exists between building legal blocks, and applying optimizations. Chapter 5 also shows how to optimize blocks using the dataflow predication model to reduce the overheads of predicate fanout and increase the amount of speculative execution within and across blocks.

Evaluation: Chapter 6 provides an evaluation of the complexity of the algorithms developed in this research by measuring the compile time spent in the individual back end phases. Chapter 6 also compares the speedup on the SPEC2000 benchmark suite for the optimizations developed in Chapter 5, and provides an evaluation of some of the design decisions made in the TRIPS ISA.

Chapter 2

TRIPS Compiler Overview

The TRIPS compiler extends the Scale retargetable compiler for C and FORTRAN. Scale is written in Java and supports the Sparc, Alpha, PowerPC, and TRIPS ISAs. Figure 2.1 shows the three major components of the compiler: the front end, the target-independent optimizer, and the target specific back ends.

Front End: The input to the Scale compiler is C89 or FORTRAN 77. Some support for C99 and gcc specific extensions is included in the C front end. After parsing, the front ends generate an abstract syntax tree (AST) in Clef [13] form. The AST can be written out to a file as C code on demand.

Target-Independent Optimizer: The abstract syntax tree is converted to a target-independent control flow graph (CFG) called Scribble. The compiler performs alias analysis and array dependence analysis on this form in preparation for optimization. The compiler can read in (or insert) profile information to guide optimization and can write the Scribble CFG out to C at any time, which is useful when debugging an optimization pass.

Command line options control the optimizations applied and their order. The compiler performs inlining, followed by loop unrolling, loop flatten-

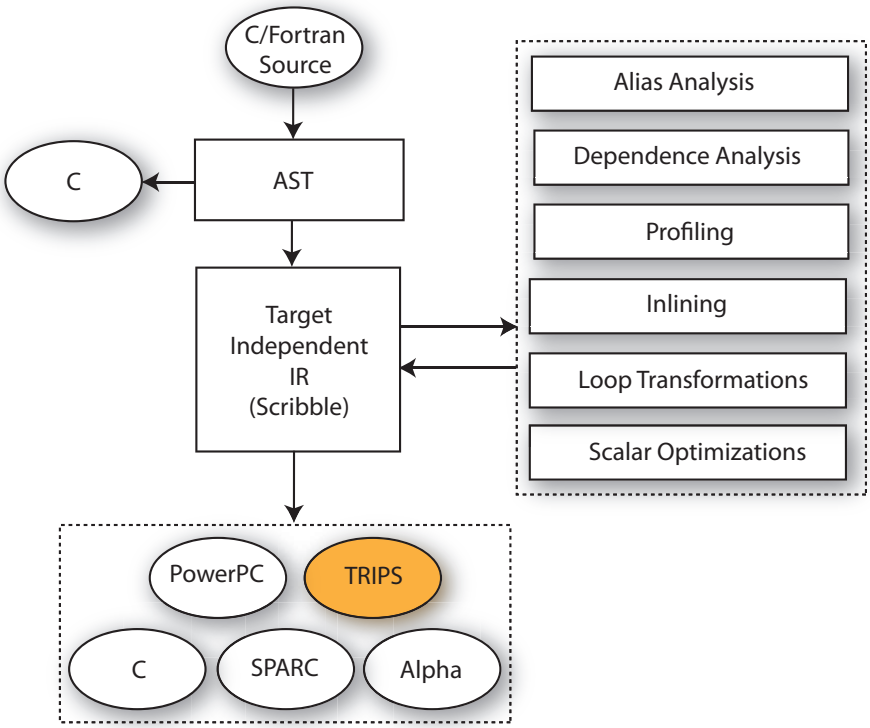


Figure 2.1: Scale Compiler Overview

ing, and loop interchange. It implements the following optimizations using static single assignment form (SSA) [23]: sparse conditional constant propagation [81], copy propagation, value numbering, loop invariant code motion, scalar replacement for array elements, and partial redundancy elimination [34]. Additionally, non-SSA versions of: global variable replacement, useless copy removal, dead variable elimination, and placing C structure fields in registers are implemented. Basic block redundant load and store elimination, and tree height reduction can also be performed in or out of SSA form.

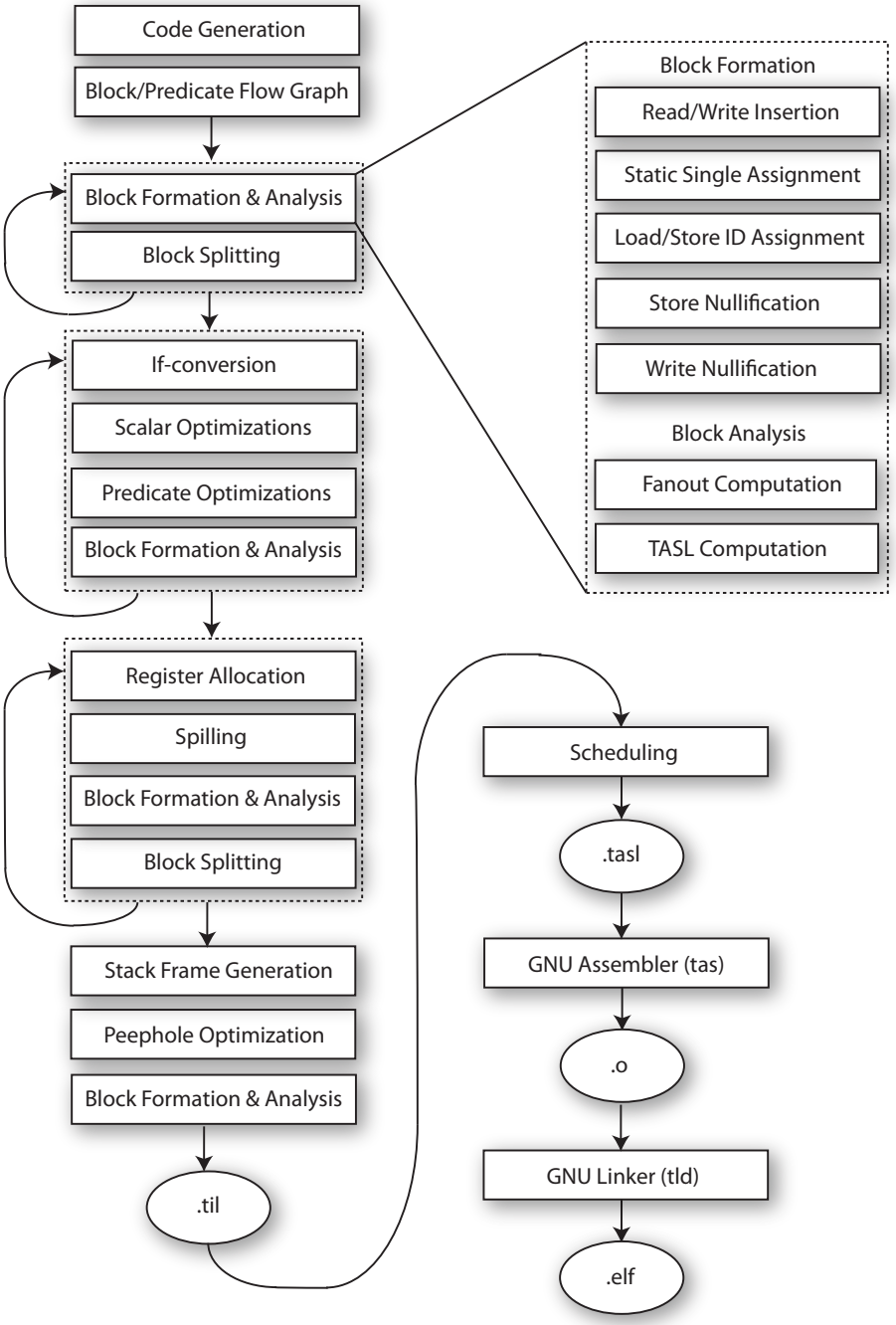


Figure 2.2: TRIPS Compiler Back End

Code Generation: After optimization, the TRIPS back end shown in Figure 2.2, generates instructions in three-address form [69] from the Scribble CFG. This is the first TRIPS-specific phase of the compiler. Code generation in an EDGE compiler differs from that of a more traditional CISC/RISC compiler because of the block based execution model. In the block based model there is no concept of a fall through branch. The compiler inserts unconditional and predicated branch instructions to make all branches explicit. For example, consider the following source program:

```
if (x > 1) {
  i++;
}
```

In a RISC architecture, if the "branch less than or equal" instruction was false, execution would fall through, and continue at the `addi` instruction following the branch.

```
L1: ble  x, 1, L2   // execution falls-through if x > 1
    addi i, i, 1
L2: ...
```

However, in the block based execution model, the compiler must create a predicate and two predicated branches. If `x > 1`, then the predicate `p1` will be true, and the processor will branch to the block beginning at label `L2`.

Otherwise, `p1` will be false, and the processor will branch to the block denoted by label `L3`. Similarly, after `i` is incremented, there must be an unconditional branch to the next block of execution.

```
L1: tgti p1, x, 1
    br_t<p1> L2
    br_f<p1> L3
L2: addi i, i, 1
    br L3            // unconditional branch to L3
L3: ...
```

Explicit branches within a block simplifies the architecture at the expense of requiring additional instructions for branching. However, since the number of branches in a block is typically small compared to the number of compute instructions, this overhead is not a limiting factor for blocks.

Block/Predicate Flow Graph Construction: The code generator produces a linearized list of instructions in three-address form. This intermediate representation is unsuitable for building and analyzing blocks since the intra-block paths of execution are obscured by predication. In this step, the compiler builds a block flow graph (the block equivalent of a control flow graph), and a predicate flow graph for each block (which makes control flow explicit within predicated code). The compiler uses this representation to progressively optimize and lower blocks until they meet the architectural constraints.

Block Formation: The next step is to transform each block into block form. In block form, compute instructions within a block target each other directly, in dataflow fashion. The compiler inserts register read and write instructions, transforms the block into dataflow form using static single assignment, nullifies register writes, assigns load-store identifiers to memory instructions, and nullifies stores. Block formation can be re-applied by any phase of the compiler. For example, the hyperblock generator applies if-conversion and combines blocks together. This merging changes the read and write instructions for a block, which changes the nulls required for write nullification. The number of memory instructions also changes, and the respective load-store identifier assignment, and store nullification. The hyperblock generator reruns the block formation algorithms as required, which is simpler than requiring every transformation in the backend to try to incrementally maintain block form at the expense of additional compile time.

- **Read and Write Insertion:** In the TRIPS ISA, compute instructions within a block cannot access the register file directly. The compiler identifies the global registers used by each block and inserts `read` instructions at the beginning of the blocks and `write` instructions at the end of the blocks.

- **Static Single Assignment:** After adding the read and write instructions to blocks, the compiler uses SSA to rename the temporary registers in each block. SSA removes any references to the global registers from

the compute instructions and replaces them with references to the `read` and `write` instructions.

- **Write Nullification:** After SSA renaming, the compiler identifies each `write` instruction that needs to be nullified and inserts any required `null` instructions.

- **Load-Store Identifier Assignment:** The compiler assigns each load and store instruction an identifier. The simplest approach is to assign a unique identifier to each memory instruction in program order. However, using the dataflow predication model, identifiers can be overlapped to increase the number of static load and store instructions that may be assigned to a block.

- **Store Nullification:** Once the LSIDs are assigned, the compiler identifies the set of stores in a block that must be nullified and inserts any required store nullification.

Block Analysis: Once in block form, the compiler analyzes each block to determine the number of machine instructions required for the block and to collect additional information required by back end transformations.

Block Splitting: The code generator may create basic blocks that are larger than the 128 instruction limit or contain more than 32 LSIDs. As a pre-pass to hyperblock formation, the block splitter analyzes all the blocks and re-forms those that violate these two constraints. For basic blocks, the

block splitter selects a location to divide the block and inserts an unconditional branch and label. Later passes may produce predicated hyperblocks, in which case the block splitter performs reverse if-conversion [9, 80]. Whenever the block splitter changes a block, the block formation algorithms must be re-applied, and the block must be re-analyzed before being checked again for a violation.

Before Block Splitting

```
L1: addi t1, t2, 1   // block has greater than 128 instructions
    ...
    subi t3, t4, 2
    ...
    br L2
L2: ...
```

After Block Splitting

```
L1: addi t1, t2, 1
    ...
    br L3            // insert unconditional branch
L3: subi t3, t4, 2   // to new label
    ...
    br L2
L2: ...
```

After code generation, predication only exists to handle branching as discussed previously. We could have eliminated the need for this block splitting phase by modifying the code generator to keep track of the constraints in the blocks. We chose though to keep the code generator focused on instruction selection, and not on discerning the TRIPS specific architectural constraints. Also, a block splitting phase is required during register allocation (to handle blocks with spill code that violate a constraint), so we were able to resuse existing code in the compiler.

Hyperblock Formation and Optimization: After code generation and block splitting, the compiler tries to increase the execution window size (for performance) by combining independent regions of code into hyperblocks through if-conversion [2]. As part of this work we developed a hyperblock generator that iteratively combines blocks using if-conversion, applies scalar and predicate optimizations to the new block, and then checks for legality with respect to the block constraints. If the block is legal, the hyperblock generator replaces the original blocks with the new block. Blocks that violate constraints are discarded and care is taken so that the hyperblock generator will not attempt to merge the same blocks again. The hyperblock generator iterates until there are no more candidate blocks to combine.

Register Allocation: After hyperblock formation, all blocks are guaranteed to be architecturally valid with respect to the block size and LSID constraints. However, blocks still reference virtual registers and may exceed the register constraints. The goal of register allocation in the TRIPS compiler

is thus to assign all virtual registers to real registers, and to ensure that no block after register allocation violates any of the block constraints. We use a modified linear scan register allocator to perform the assignment. If a block requires more than the 32 reads or 32 writes, the allocator spills (inserting load and store instructions as required). Since spilling increases the number of instructions and the number of LSIDs in a block, the block splitter analyzes each block with spills to determine if either the block size or LSID constraints are violated. If no constraint is violated then register allocation is complete and all blocks adhere to the block constraints. However, if there is a violation due to spilling, all spill code is removed and the block splitter divides the blocks with violations into smaller blocks, and register allocation is repeated. Eventually register allocation will terminate since the block splitter is creating smaller blocks with fewer instructions.

Stack Frame Generation: Once real registers are assigned the compiler generates the stack frame. Since the instructions in the stack frame are always executed, the compiler can track the block constraints as instructions are generated. If the limit for any constraint is reached then the compiler generates a new block and continues creating the stack frame. For simplicity, the prologue and epilogue are initially generated in their own blocks. Then the compiler attempts to combine the prologue and epilogue with their successor and predecessor blocks respectively [67].

Peephole Optimization: After stack frame generation, all code has been generated, hyperblocks have been formed, and all block constraints are

guaranteed to be satisfied. The compiler applies peephole patterns to clean up the code and applies the block formation algorithms again to finalize the blocks. The blocks are also analyzed so that statistics about the final code can be generated if requested by the user.

Scheduling, Assembly, and Linking: Finally, the compiler writes the architecturally valid blocks to a file in TRIPS Intermediate Language (TIL) [70] form. TIL is a RISC-like intermediate representation that does not consider physical placement of instructions. The scheduler [52] reads in the TIL, maps TIL instructions to execution tiles, and writes out the resulting scheduled blocks in TRIPS Assembly Language (TASL) [82]. After scheduling, the TRIPS assembler (tas) and linker (tld) which are based upon the GNU assembler and linker are used to create an ELF binary [83].

Chapter 3

Forming Blocks

The block is the unit of work in an EDGE architecture. To effectively compile to block-based targets, we must develop a representation for blocks in the compiler, along with compiler algorithms to analyze and optimize instructions in block form. To simplify the discussions in this chapter, we assume that we are compiling to an architecture with no block constraints. In Chapter 4 we will describe the algorithms necessary for dealing with a fixed number of block constraints such as those imposed by the TRIPS prototype processor. This chapter is structured as follows: first we describe the representation for blocks in the compiler. Then, we describe how to compute liveness over this representation as the algorithms for building block form rely on knowing what values are live. Finally, we describe the algorithms for transforming TIL into block form.

Figure 3.1 depicts both a C routine to compute Fibonacci numbers and the corresponding three-address TIL code after instruction selection. At this stage in the compiler, all instructions reference virtual registers. We require a representation that makes the block boundaries explicit, and allows the compiler to reason about the flow of control both between blocks and within

```
long fib(unsigned long n) {
  if (n <= 1) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

```
fib:   mov     t128, t3
       tleui   t132, t128, #1
       bro_t<132>  fib$4
       bro_f<132>  fib$1
fib$1: subi    t3, t128, #1
       enterb  t2, fib$2
       callo   fib
fib$2: mov     t130, t3
       subi    t3, t128, #2
       enterb  t2, fib$3
       callo   fib
fib$3: mov     t131, t3
       add     t128, t130, t131
       bro     fib$4
fib$4: mov     t3, t128
       ret     t2
fib$5: bro     fib$4
```

Figure 3.1: The C source to compute a Fibonacci number and corresponding intermediate TIL code.

blocks. A block corresponds to a region of code with a single unpredicated entry, and either a single unpredicated exit or multiple predicated exits.[1] In TIL, a block corresponds to the instructions between two consecutive labels. For example, the label `fib` in Figure 3.1 begins a block that encompasses all the instructions up to the label `fib$1`. By examining the labels and branches in the TIL, the compiler builds a *block flow graph* (BFG), which is equivalent to a control flow graph for blocks.

[1]Blocks are not restricted by the EDGE architecture to being single entry. If inter-block predicates were allowed, then blocks could have multiple predicated entrances. However, in this work we only consider single entry blocks.

The block flow graph makes the flow of control between blocks explicit, but does not contain any information about the intra-block paths of execution that exist in predicated code. For example, the block rooted at `fib` has two possible paths of execution corresponding to the two branch instructions (`bro`) predicated on opposite conditions. Previous work on compilers for predicated VLIW machines introduced a graph based representation for predicated code, the *predicate flow graph* [6]. In the predicate flow graph, instructions that create predicates are analogous to branch instructions, and the first instruction to use a predicate is analogous to a label. Combining these two representations we arrive at the following three-level hierarchical graph to enable transformations on predicated code:

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

Predicate Flow Graph (PFG): A node in the PFG 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 block. To provide a convenient location to place register write instructions, we add an empty, unpredicated, final merge node to every PFG (if one does not already exist). The PFG is a directed acyclic graph.

Predicate Block: Each predicate block is a basic block of instruc-

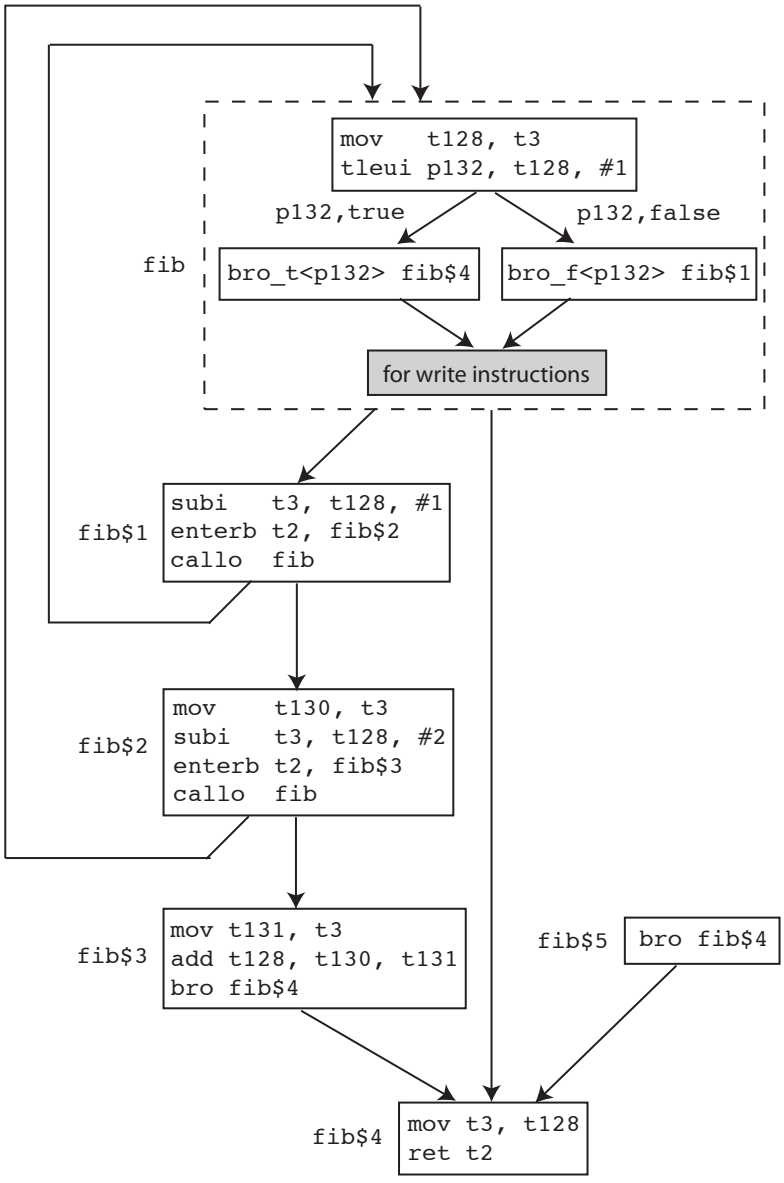


Figure 3.2: Complete block flow graph for the Fibonacci example in Figure 3.1. All blocks contain a single predicate block except for the block enclosed in dashed lines.

tions with uniform predication; either no instructions are predicated, or all are predicated on the same predicate.

Figure 3.2 shows a complete block flow graph for the Fibonacci example in Figure 3.1. The predicate flow graph for block *fib* is enclosed in dotted lines since it contains multiple predicate blocks. There is an empty predicate block in this PFG as a place to insert write instructions. Every other block in the block flow graph contains only a single predicate block.

3.1 Building the Block Flow Graph

The compiler uses a two-pass algorithm to build the block flow graph for a procedure, starting with the linear intermediate representation (IR) of TIL instructions immediately after code generation (Figure 2.2). On the first pass an empty block is created for each label in the TIL and the block is added to a hash table with the label as the key. For the Fibonacci example in Figure 3.2, six blocks are created (`fib`, `fib$1`, etc).

The second pass adds edges between blocks by examining the instructions that cause control flow between them. These instructions consist of conditional and unconditional branches, and function calls since a function cannot return into the block containing the original call. An edge is added between the block containing the control flow instruction and the target block of the instruction. We traverse the instructions in a forward pass, and whenever we see a label, we perform a hash table lookup using the label as the key. The block that is returned from the lookup is then used as the current block. As

we encounter branch and call instructions, we retrieve their target blocks from the hash table, and add an edge in the graph between the current block and the target block. After the block flow graph is created, the compiler creates the predicate flow graphs for each block.

3.1.1 Removing Unreachable Blocks

As an artifact of code generation, there may be unreachable blocks in the block flow graph. Even though these blocks will never execute, eliminating them can reduce code size. In the Fibonacci example of Figure 3.1, `fib$5` corresponds to the block following the `if-then-else`. Since both sides of the `if` contain return statements, block `fib$5` is unreachable.

The compiler removes unreachable blocks from the block flow graph using the following algorithm:

1. Tag every block in the BFG with a 0.
2. Perform a depth first search from the root of the BFG, tagging every block with a 1.
3. Remove any block in the BFG whose tag is not 1.

This algorithm can also provide additional information for later when we are generating the stack frame. Consider this function which contains a loop that never terminates:

```
int foo() {
   for (;;)
      continue;
   return 1;
}
```

Since the block that contains the return instruction is unreachable the function will never return to the caller and the compiler does not need to generate the epilogue of the stack frame for the function. When removing unreachable blocks, the compiler can note a function does not return if the tag for the block containing the return is set to 0.

3.2 Building the Predicate Flow Graph

Building the predicate flow graph is similar to the traditional control-flow graph construction algorithm of identifying "leaders and labels" [21]. In the PFG, a leader corresponds to an instruction that creates a predicate, and a label corresponds to the first instruction that uses a predicate. Splits are created in the PFG immediately after instructions that define predicates, and merges are created in the graph immediately preceding unpredicated instructions. Before the compiler can construct the PFG, it must identify all the instructions that define predicates.

TRIPS uses a dataflow predication model described in Section 1.3. In this model any instruction that produces a value can instead produce a pred-

icate. Predicates are routed over the operand network to their dependent instructions just like any other value. Identifying the predicates in a block is done with a backward pass over the instructions. For each instruction, the instruction's predicate field is examined, and a bit vector records any temporary register name used as a predicate. Next, the instruction's destination register is checked. If the destination register is set in the bit vector, then the instruction defines a predicate and is marked. When the compiler reaches a label in the backward pass, it clears the bit vector since predicates are not live across block boundaries.

The algorithm in Figure 3.3 constructs the PFG for a given block. We rely on the fact that the PFG is a directed acyclic graph to simplify the construction. The compiler first creates an unpredicated predicate block[2] for the entry to the PFG and appends the label as the first instruction. Then for each instruction in a forward pass, the compiler destructively moves the instruction from the linear IR to its predicate block. The predicate block for an instruction is determined by the routine `GET_BLOCK()`, which uses the predicate for the instruction to lookup the predicate block in two hash tables: one hash table contains all the predicate blocks for "true" predicates, the other for "false" predicates. The retrieval can never fail because predicates can only be used after they are defined (and their associated predicate blocks).

When an instruction defines a predicate, `ADD_EDGES()` inserts the re-

[2]An unpredicated predicate block is a basic block. However, we make the distinction to be clear that we are discussing predicate blocks in the PFG.

```
function CREATE_PREDICATE_FLOWGRAPH(Instruction: First)
returns PredicateBlock
  Prev: First
  Entry: new PredicateBlock
  add First to Entry
  foreach instruction I beginning with First.next
     Prev.next = null
     if I is label
       break
     endif
     B = GET_BLOCK(I)
     add I to B
     if I defines a predicate
       ADD_EDGES(B, I)
     endif
     Prev = I
  endfor
  if B is predicated
    INSERT_MERGE()
  endif
  return Entry
endfunction CREATE_PREDICATE_FLOWGRAPH
```

Figure 3.3: Algorithm to transform a linear sequence of instructions into a predicate flow graph.

quired edges in the graph. The number of edges inserted depends on the predicate being created. For example, a test instruction creates a bi-directional predicate that evaluates to either true or false. In this case, the two predicate blocks representing the true and false paths of the predicate are retrieved from their respective hash tables (the compiler creates the predicate blocks the first time the predicate is used), and edges are inserted to connect them with the

predicate block containing the test instruction.

If an instruction is unpredicated, a new unpredicated predicate block is created, and inserted as a merge in the PFG by adding edges to all the leaf predicate blocks. The compiler saves the last predicate block used to append an instruction. Before performing any hash table lookups or creating a new unpredicated predicate block, the compiler compares the predicate for the saved predicate block with the predicate for the instruction to be appended. This comparison ensures that consecutive sequences of unpredicated instructions are appended to the same predicate block and reduces the number of hash table lookups for predicated code. Finally, an unpredicated merge is added to the PFG if the PFG does not already end in one.

3.3 Computing Liveness

Fundamental to any compiler is the computation of live variables in a program. At the granularity of a block, liveness is the set of variables live-in to or live-out of a block for all paths of execution through the block. For optimizations at a block level such as register allocation and inserting read and write instructions, this definition of liveness where we treat blocks as large instructions, is sufficient. However, some intra-block optimizations require knowledge about live variables between predicate blocks and so we account for both uses. To compute liveness for blocks, we extend backwards iterative dataflow analysis. At a high level the algorithm is as follows:

1. Compute the initial $use()$, $def()$, $in()$, $out()$ sets for each predicate block in every PFG in the BFG. Care must be taken when computing the initial $out()$ sets to account for precolored registers[3] that are always live-out of a block. The complete algorithm is given in Figure 3.5.

2. Iterate backwards. In our compiler, we only build local PFGs for each block. During liveness analysis we construct a global PFG by building a table that maps the label of a block to the predicate blocks that branch to the label. As we iterate backwards, whenever we reach the label corresponding to the entry to a PFG, we use this table to determine the next predecessor predicate block to process.

3. When there are no more changes, we compute the $use()$, $def()$, $in()$, $out()$ sets for a block from the sets that have been computed for each predicate block in the block's PFG.

To understand how liveness is computed, we examine the standard iterative method over the BFG in Figure 3.4. The initial $use()$, $def()$ sets for this flow graph are as follows:

[3]A precolored register is a virtual register that has already been assigned to a machine register.

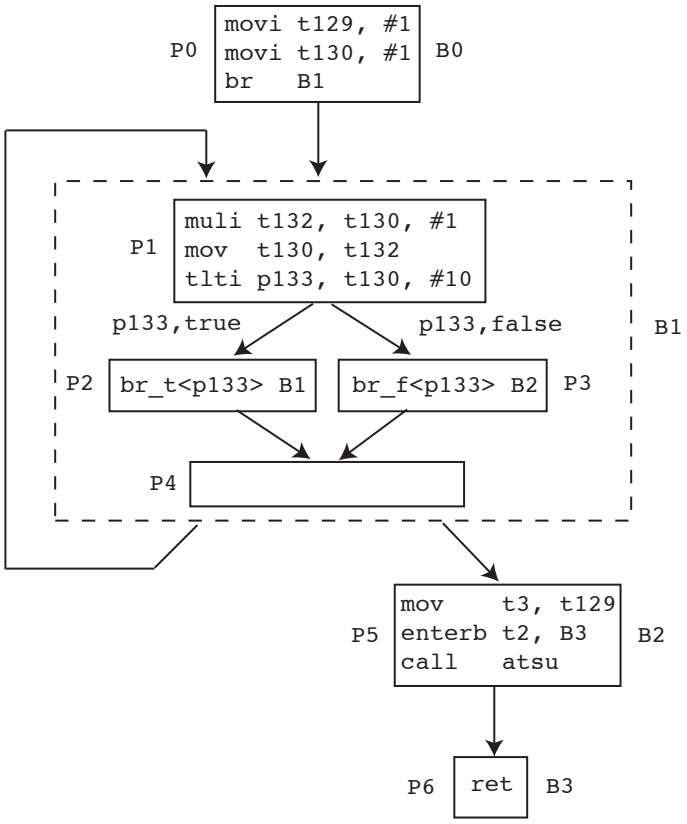


Figure 3.4: Example block flow graph for computing live variables.

$use(P0) = \emptyset$ $def(P0) = \{t129, t130\}$
$use(P1) = \{t130\}$ $def(P1) = \{t130, t132, p133\}$
$use(P2) = \{p133\}$ $def(P2) = \emptyset$
$use(P3) = \{p133\}$ $def(P3) = \emptyset$
$use(P4) = \emptyset$ $def(P4) = \emptyset$
$use(P5) = \{t129\}$ $def(P5) = \{t2, t3\}$
$use(P6) = \emptyset$ $def(P6) = \emptyset$

The final $in()$, $out()$ sets for each predicate block are:

$in(P0) = \emptyset$ $out(P0) = \{t129, t130\}$
$in(P1) = \{t129, t130\}$ $out(P1) = \{t129, t130, p133\}$
$in(P2) = \{t130, p133\}$ $out(P2) = \{t130\}$
$in(P3) = \{t129, p133\}$ $out(P3) = \{t129\}$
$in(P4) = \emptyset$ $out(P4) = \emptyset$
$in(P5) = \{t129\}$ $out(P5) = \emptyset$
$in(P6) = \emptyset$ $out(P6) = \emptyset$

There is a problem with the treatment of precolored registers in the example. In block `B2`, both `t2` and `t3` are precolored since they are arguments to the function call and must be written to the global register file. It would be incorrect for the register allocator to use `t2` for instance, to hold a value between block `B2` and `B3`. The solution to this problem is to add defined precolored registers to the $out()$ sets of their containing predicate blocks when computing the initial sets.

Once we compute liveness for each predicate block, we compute the information for each block in the BFG (Figure 3.6). The live-ins to a block are the same as the live-ins to the predicate block that is the entry to the block's PFG. The live-outs are the union of the live-outs from every predicate block that is an exit from the block. For example, the entry to block `B1` is `P1`

```
procedure COMPUTE_INITIAL_SETS
  foreach PredicateBlock in BFG
    In: PredicateBlock.In
    Out: PredicateBlock.Out
    Uses: PredicateBlock.Uses
    Defs: PredicateBlock.Defs
    foreach Instruction in PredicateBlock
      foreach Src in Instruction
        if Src not in Defs then
          add Src to In
        endif
        add Src to Uses
      endfor
      foreach Dest in Instruction
        add Dest to Defs
        if Dest is precolored then
          add Dest to Out
        endif
      endfor
      if Instruction is call then
        add Instruction.Uses to Out
      endif
    endfor
  endfor
endprocedure COMPUTE_INITIAL_SETS
```

Figure 3.5: Computing the initial sets for liveness on the BFG.

```
procedure COMPUTE_FINAL_SETS
  foreach Block in BFG
    In: Block.In
    Out: Block.Out
    Use: Block.Use
    Def: Block.Def
    Entry: Block.PFG.Entry
    foreach PredicateBlock in Block
      Def OR PredicateBlock.Def
      Use OR PredicateBlock.Use
      if Block contains a branch then
        Out OR PredicateBlock.Out
      endif
    endfor
    In = Entry.In
  endfor
endprocedure COMPUTE_FINAL_SETS
```

Figure 3.6: Computing the finals sets for liveness on the BFG.

and there are two exits–P2 and P3–resulting in,

$$in(B1) = \{t129, t130\} \quad out(B1) = \{t129, t130\}$$

The interplay of predicate block P4 and the analysis deserves some explanation. Recall that this predicate block is *s*tructural and added to the PFG during construction to hold write instructions. At all other times this block is empty and thus has no effect on the dataflow analysis. In fact, by building the global PFG in step 2, the backwards analysis moves from P5 to P3 and from P1 to P2 skipping P4. Our algorithm does include P4 when computing the initial sets, which is sufficient to compute the correct live-ins

and live-outs when it is non-empty (such as after stack frame generation).

3.4 Read and Write Insertion

The first step in block formation is to add the register read and write instructions to a block. Before register allocation read and write instructions reference virtual registers, and after register allocation they reference physical registers. The compiler performs standard live variables analysis on the BFG. For each register live-in to a block (and used by an instruction in the block), the compiler inserts a `read` in the predicate block that is the entry to the PFG. For each register live-out of a block (and defined by an instruction in the block), the compiler inserts a `write` in the predicate block that is the final unpredicated merge created when forming the PFG. The compiler uses the same register for both the source and destination operands to facilitate renaming when entering SSA form. This leads to the following dataflow equation, where `i` corresponds to a block in the BFG:

$$reads(i) = in(i) \cap use(i)$$
$$writes(i) = out(i) \cap def(i)$$

Computing liveness on the blocks in Figure 3.7(a), results in the following $use()$, $def()$ sets:

$$use(B0) = \emptyset \qquad def(B1) = \{t9\}$$
$$use(B1) = \emptyset \qquad def(B1) = \{t8\}$$
$$use(B2) = \{t8, t9\} \quad def(B1) = \{t2, t3\}$$

```
B0: movi   t9, #4
    br     B1
B1: movi   t8, #150
    br     B2
B2: subi   t3, t9, t8
    enterb t2, B2
    call   out
```

(a)

```
B0: movi   t9, #4
    br     B1
    write  t9, t9
B1: movi   t8, #150
    br     B2
    write  t8, t8
B2: read   t9, t9
    read   t8, t8
    sub    t3, t9, t8
    enterb t2, B2
    call   out
    write  t2, t2
    write  t3, t3
```

(b)

Figure 3.7: Example of adding read and write instructions to blocks.

and the following $in()$, $out()$ sets:

$$\begin{array}{ll} in(B0) = \emptyset & out(B0) = \{t9\} \\ in(B1) = \{t9\} & out(B1) = \{t8, t9\} \\ in(B2) = \{t8, t9\} & out(B2) = \{t2, t3\} \end{array}$$

Figure 3.7(b) gives the blocks after adding the read and write instructions:

$$\begin{array}{ll} read(B0) = \emptyset & write(B0) = \{t9\} \\ read(B1) = \emptyset & write(B1) = \{t8\} \\ read(B2) = \{t8, t9\} & write(B2) = \{t2, t3\} \end{array}$$

```
B0: movi   t19, #4
    br     B1
    write  t9, t19
B1: movi   t18, #150
    br     B2
    write  t8, t18
B2: read   t29, t9
    read   t28, t8
    sub    t13, t29, t28
    enterb t12, B2
    call   out
    write  t2, t12     ; t2 is precolored
    write  t3, t13     ; t3 is precolored
```

Figure 3.8: The example from Figure 3.7(b) after SSA renaming.

3.5 Building Static Single Assignment Form

The next step after read and write insertion is to transform the block into static single assignment form, which serves two purposes:

1. All references to global registers are removed from the compute instructions within the block, leaving only the `read` and `write` instructions to access them.

2. Temporary registers within a block are renamed to put the block in a pure dataflow form and facilitate conversion to target form.

Figure 3.8 illustrates (1) by showing the example in Figure 3.7(b) after SSA renaming. However, (2) requires a more detailed explanation. Figure 3.9(a) contains a block that has not been renamed. Assume for a moment

```
read t7, g10                    R0: read I0, I2 [G10]
addi t8, t7, #3                 I0: addi #3, I1, I3      ; wrong
mov  t9, t8                     I1: mov  I3
slli t8, t7, #4                 I2: slli #4, I3
tgt  t2, t8, t9                 I3: tgt  I4, I5
br_t<t2> ogai                   I4: br   ogai
br_f<t2> mori                   I5: br   mori

     (a)                                (b)
```

Figure 3.9: A block (a) without SSA renaming and (b) after (incorrect) conversion to target form.

that the temporary registers are actually physical register names and not edges in a data dependence graph. Even though `t8` is defined twice, once by the `addi` and again by the `slli`, in a processor that holds operand values in registers, this code will execute correctly. However, in the EDGE block-based dataflow execution model, the temporary operands actually represent dataflow edges. When the compiler converts to target form, it will target temporaries with the same name to the same instructions. In Figure 3.9(b), the `addi` instruction would wrongly target both the `mov` and the `tgt` instructions, which will lead to undefined behaviour at runtime. The `addi` should only target the `mov`. SSA renaming solves this problem by assigning unique names to temporaries.

We use a standard implementation of SSA [14] with two exceptions. The first exception deals with our treatment of copy folding when entering SSA form. The second exception is required to maintain the predicate flow graph.

3.5.1 Copy Folding

When entering SSA form it is often desirable to perform copy folding to remove unneeded copy instructions. However, in block form not all copies can be folded. Consider the case when there is a single `mov` instruction that references two global registers:

```
L1: mov g3, g4
    br  L2
```

Before entering SSA, the compiler inserts `read` and `write` instructions and then uses SSA to rename the block. If we perform copy folding on this example the `mov` instruction becomes a `nop`.

```
L1: read  t100, g4
    nop
    br    L2
    write g3, t100
```

A problem arises if we want to remove the `read` and `write` instructions from the block (perhaps for additional optimization). We would like to rename the original `mov` that referenced the global registers, then remove the `read` and `write`. But in this example, the `mov` instruction has been deleted by copy folding leaving nothing to rename. Our solution is to fold only copies that do not reference values live across block boundaries.

3.5.2 Maintaining the Predicate Flow Graph

Each predicate block in the PFG has an associated predicate (unless the predicate block is unpredicated) that is defined by an instruction. After SSA renaming, the predicate blocks reference the pre-renaming predicates and instructions no longer map correctly to the PFG. Optimizations that move instructions between predicate blocks will not be able to use the PFG to determine the moved instruction's predicate. An extra step in SSA renaming renames the predicate for a predicate block to maintain the PFG. After renaming all the instructions in a predicate block, SSA uses the rename stack to rename the predicate for the predicate block just as it would for an instruction.

3.6 Write Nullification

The constant output constraint requires that a block produce the same set of register writes for all paths of execution through the block to simplify the hardware detection of block completion. In the absence of predication there is only one path of execution through a block and this constraint is trivially satisfied, but in predicated code there may be multiple paths of execution each producing a different set of register writes. The compiler must ensure that for every path that writes a global register, every other path writes the same register. The compiler uses SSA to identify the paths that are missing register writes.

In Figure 3.10(a), there are two paths of execution: on the $\{p1, true\}$ path `t4` is produced and written to a global register, however, on the $\{p1, false\}$

```
read t2, t1
read t3, t22
tgti p1, t2, #0
ld_t<p1>   t4, 0(t3)
movi_f<p1> t6, #12
sd_f<p1>   8(t3), t6
phi   t8, t4, t16 ; undefined
br    L2
write t16, t8
```

(a)

```
read t7, t16  ; added
read t2, t1
read t3, t22
tgti p1, t2, #0
ld_t<p1>   t4, 0(t3)
movi_f<p1> t6, #12
sd_f<p1>   8(t3), t6
phi   t8, t4, t7
br    L2
write t16, t8
```

(b)

Figure 3.10: A block in SSA form (a) that contains a write with an undefined operand and (b) defining the write by reading in the original register value.

path no value is produced and sent to the write. After transforming into SSA form, the paths that are missing definitions will have `phi` instructions with operands that have not been renamed, since they were not defined. For example the `phi` in Figure 3.11(a) contains an undefined operand `t16`. The compiler can provide a definition for these operands using one of the following techniques:

1. The block can read the registers for the writes that are missing definitions and write the original register values back out as in Figure 3.10(b). When the compiler inserts `read` and `write` instructions, it simply inserts reads for all the live-ins and live-outs of a block giving every write a corresponding read. Any unneeded register reads will be removed from the block by dead code elimination later. This solution generally increases

the register pressure for a block (unless a register was already being read), but can be used to generate correct code quickly.

2. The compiler can perform *write nullification* by inserting `null` instructions that define a `null` value for the register writes on the undefined paths. The `null` value signals to the architecture that no value will be produced for the register write.

To perform write nullification, the compiler searches the use-def chains for all `write` instructions. If the search finds a `phi` instruction with an operand that does not have a definition, the operand is nullified by inserting a `null` instruction on the path missing the definition, which ensures that all paths produce the same set of register writes. However, consider the case where a block is already reading and writing the same global register as in Figure 3.11(a). On the $\{p1, false\}$ path `t24` is being read and written without any change to the original value. When searching a `write's` use-def chains, if a `read` instruction is encountered, the `read's` source operand is compared against the `write's` destination operand. If they match, the compiler inserts a `null` on this path. Later on, if there are no uses of the `read` instruction, dead code elimination will remove it from the block.

3.7 Load and Store Identifier Assignment

To guarantee sequential memory semantics, the compiler assigns a unique ordering identifier (LSID) to each load and store in a block, and the

```
read t2, t1
read t3, t22
read t12, t24 ; same as write
tgti p1, t2, #0
ld_t<p1>   t4, 0(t3)
movi_t<p1> t11, #7
movi_f<p1> t6, #12
sd_f<p1>   8(t3), t6
phi   t8, t4, t16 ; undefined
phi   t9, t11, t12
br    L2
write t16, t8
write t24, t9
```

(a)

```
read t2, t1
read t3, t22
tgti p1, t2, #0
ld_t<p1>   t4, 0(t3)
movi_f<p1> t6, #12
sd_f<p1>   8(t3), t6
null_f<p1> t7 ; added
null_f<p1> t12 ; added
phi   t8, t4, t7
phi   t9, t11, t12
br    L2
write t16, t8
write t24, t9
```

(b)

Figure 3.11: A block in SSA form (a) before write nullification and (b) after write nullification.

microarchitecture ensures the same results as if loads and stores were executed sequentially in LSID order [63]. The compiler can satisfy this constraint by assigning a unique identifier to every load and store in reverse post order as in Figure 3.12(a). With this assignment, the number of load-store identifiers required for a block is equal to the total number of load and store instructions in the block. However, the compiler can improve upon this assignment since it is free to reuse LSIDs if it can guarantee that only one load or store will ever fire for a given identifier. In other words, load and store instructions on mutually exclusive paths of execution can share identifiers with the restriction that a load and store instruction can never have the same identifier since the hardware block termination logic uses store identifiers to detect block comple-

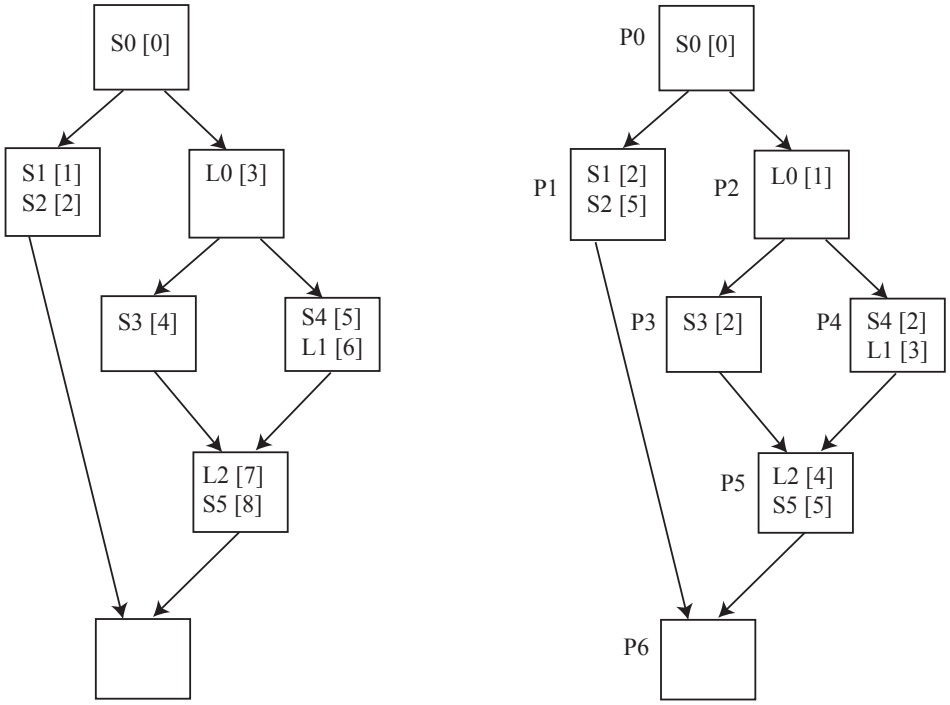


Figure 3.12: Load-store identifier assignment using (a) a maximal assignment that gives every load and store a unique identifier and (b) a policy that shares identifiers between stores and assigns a unique identifier to every load.

tion. If the compiler reuses LSIDs, it can include more loads and stores in a block, increasing the block size.

Our policy for assigning LSIDs between memory instructions is based on the following observations:

1. Every path of execution through a block must produce the same set of store identifiers. When a store identifier appears on one path but not another, the compiler must insert additional instructions to nullify the

missing store identifier. Therefore, a policy that minimizes the number of store identifiers in a block will also minimize the number of overhead instructions that must be inserted for store nullification.

2. Inside a block we often want to unpredicate load instructions so they can execute speculatively to hide latency. However, in the TRIPS microarchitecture, loads that share LSIDs cannot execute speculatively since two or more load instructions may execute with the same LSID, in violation of the block constraints. Therefore, we prefer a policy that gives the compiler the maximum amount of freedom when determining what load instructions to speculatively execute.

Both conditions can be satisfied by an algorithm that *minimizes the number of LSIDs assigned to stores and maximizes the number of LSIDs assigned to loads*. To implement this algorithm the compiler uses a worklist that contains the predicate blocks to process along with the next memory instruction in each predicate block to be assigned. If any one of the instructions is a load, the compiler assigns the load the next LSID. If *every* one of the instructions are stores, the compiler assigns all the stores the same LSID. Whenever an LSID is assigned, the compiler updates the next instruction to be processed to the next load or store in the predicate block. When the compiler reaches the end of a predicate block it places any successor predicate blocks on the worklist if all the predecessors of a successor have been visited and assigned.

The compiler's LSID assignment algorithm is best explained using the

example from Figure 3.12(b). The worklist initially contains {*P0,S0*} and *S0* is assigned LSID 0. The two successor predicate blocks *P1* and *P2* are then added to the worklist since their predecessor *P0* has been processed: {{*P1,S1*},{*P2,L0*}}. *L0* is assigned LSID 1 and the successor predicate blocks *P3* and *P4* are added to the worklist: {{*P1,S1*},{*P3,S3*},{*P4,S4*}}. Now every instruction on the worklist is a store and they are assigned LSID 2. Since the successor predicate block of *P3* has a predecessor still being processed no new predicate blocks are added to the worklist: {{*P1,S2*},{*P4,L1*}}. *L1* is assigned LSID 3 and now all the predecessors of *P5* are complete so it can be added: {{*P1,S2*},{*P5,L2*}}. *L2* is then assigned LSID 4 and finally all the remaining instructions are stores that are assigned LSID 5. This algorithm provably minimizes the store identifiers because it is the same problem as typed fusion [33].

3.8 Store Nullification

The compiler must guarantee that all paths of execution produce the same set of store identifiers to satisfy the constant output constraint: once the block produces all its outputs (register writes, stores, and one branch), it completes. For every store identifier in a block, the compiler identifies the missing identifiers on each path and inserts a *nullified store*–a store with a null instruction as its operand–for the missing identifiers. Similar to write nullification, the `null` value signals to the architecture that a store will not be produced for an identifier. A nullified store takes the form:

```
null t3
sd_t<p1> 0(t3), t3 [2]
```

We call the nullified store a "dummy store", as its only purpose is to receive the null value. The instruction overhead for nullification excludes the compiler from adding other, more useful instructions, to blocks.

Computing the set of store identifiers to be nullified can be framed as a backwards dataflow problem on the PFG. For each predicate block the compiler computes the set of store identifiers $sin()$ and $sout()$. The set of identifiers $sout()$ of a predicate block is the union of the identifiers of all its successor predicate blocks. The set of identifiers $sin()$ to a predicate block is the union of the $sout()$ set plus any store identifiers defined in the predicate block. After computing all the $sin()$ and $sout()$ sets the compiler only has to examine the split points (predicate blocks with more than one successor) and their successors in the PFG to determine which identifiers are missing. Any store identifier in the split point must also be in the successor or the compiler must insert a null in the successor:

$$nulls(succ) = sout(split) - sin(succ)$$

To determine the missing store identifiers for the PFG in Figure 3.13, we first compute $sin()$ and $sout()$ for each predicate block starting with *P6* and moving backwards to *P0*:

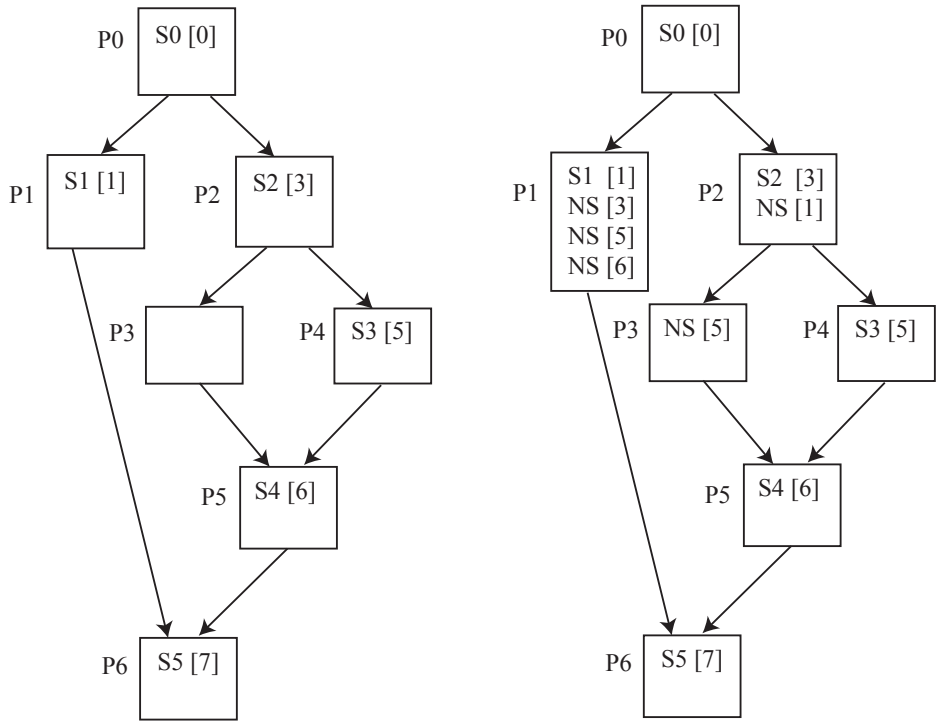


Figure 3.13: A predicate flow graph before and after store nullification (S = store, NS = nullified store).

$$
\begin{array}{ll}
sin(P6) = \{7\} & sout(P6) = \emptyset \\
sin(P5) = \{6,7\} & sout(P5) = \{7\} \\
sin(P4) = \{5,6,7\} & sout(P4) = \{6,7\} \\
sin(P3) = \{6,7\} & sout(P3) = \{6,7\} \\
sin(P2) = \{3,5,6,7\} & sout(P2) = \{5,6,7\} \\
sin(P1) = \{1,7\} & sout(P1) = \{7\} \\
sin(P0) = \{0,1,3,5,6,7\} & sout(P0) = \{1,3,5,6,7\}
\end{array}
$$

Then we examine the split points *P0* and *P2* to solve for the missing store identifiers:

$$nulls(P1) = sout(P0) - sin(P1) = \{3, 5, 6\}$$
$$nulls(P2) = sout(P0) - sin(P2) = \{1\}$$
$$nulls(P3) = sout(P2) - sin(P3) = \{5\}$$
$$nulls(P4) = sout(P2) - sin(P4) = \emptyset$$

Next, we insert null stores in *P1* with store identifiers {3,5,6}, in *P2* with {1}, and in *P3* with {5}.

3.9 Leaving SSA Form

The last step in block formation is to remove the internal `phi` instructions by transforming the block out of SSA form. Up until this point, `phis` have been used to gate operands and maintain the dataflow semantics of the block. Take for example the `phi` instruction in Figure 3.14(a). Here the `phi` is selecting between the value of two operands: `t2` and `t6`. The problem with simply removing this `phi` through renaming is illustrated by the load instruction in Figure 3.14(b) . Since the load will execute unconditionally, the `write` will receive a value regardless of which path of execution is taken within the block. There is no problem for the $\{p1, true\}$ path since the `write` should receive the value produced by the load. However, on the $\{p1, false\}$ path, both the load and `slli` instructions will produce a value that is sent to the `write`. The original `phi` instruction was used to gate the operands and select among them, and the compiler must preserve these semantics when removing the `phi`. To solve this problem when leaving SSA, form a predicated `mov` instruction is inserted into the predecessor predicate block to gate the operand as in Figure 3.14(c).

```
read t3, t22              read t3, t22              read t3, t22
ld   t2, 0(t3)            ld   t8, 0(t3) ; error    ld   t2, 0(t3)
tgti p1, t2, #0           tgti p1, t2, #0           tgti p1, t2, #0
slli_f<p1> t6, t2, #4     slli_f<p1> t8, t2, #4     mov_t<p1>  t8, t2
phi  t8, t2, t6           br   L2                   slli_f<p1> t8, t2, #4
br   L2                   write t16, t8             br   L2
write t16, t8                                       write t16, t8

     (a)                       (b)                       (c)
```

Figure 3.14: A block (a) in SSA form, (b) after transforming (incorrectly) out of SSA, and (c) after transforming (correctly) out of SSA.

For `phi` instructions that define predicates, there is no need to insert a `mov` instruction. Therefore, we always rename `phi` instructions that define predicates. We defer the renaming until after all phis (non-predicate and predicate defining) have been removed from a block. Then, we choose one of the phi's operands and use this temporary register to rename all the instructions in the block that define any of the operands referenced by the `phi`. At this time we also rename the predicates for each predicate block to maintain the PFG.

3.10 Related Work

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 [4, 25], since

```
PredicatePhis: List of phi instructions that define predicates

procedure REMOVE_PHIS(PredicateBlock: PB)
  foreach Instruction I in PB
    if I is not a Phi instruction then
       break ; phis always come first
    endif
    if I defines a predicate then
      add I to PredicatePhis
    else
      foreach SrcOperand Src in I
        if RENAME_PHI_OPERAND(Src) returns false
          insert Copy instruction in PB.Predecessor
        endif
      endfor
    endif
    remove I from PB
  endfor
  foreach PredicateBlock PBD in DominatorTree of PB
    REMOVE_PHIS(PBD)
  endfor
endprocedure REMOVE_PHIS
```

Figure 3.15: Algorithm to remove `phi` instructions when transforming out of SSA form.

dataflow machines limited the programming model to a functional one where programs cannot produce multiple memory values to the same location, relieving the compiler and architecture of the memory disambiguation problem.

There has been recent work on compilers for dataflow-like architectures similar to TRIPS [12, 15, 17]. The most closely related is the CASH compiler that targets a substrate called ASH (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, are applicable to TRIPS. 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 [73] 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 [72, 73], 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 [52].

Previous work on compilers for predicated VLIW machines introduced the predicate flow graph (PFG) [6]. In this work, like ours, predicates are treated as intra-block values, which simplifies the construction of the PFG. We give an alternate construction algorithm for the PFG, and unlike previous work, use the PFG as the back end's internal representation for lowering and optimizing predicated blocks.

3.11 Summary

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 for block temporaries. 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 block; one that permits the compiler to form large, full blocks, without requiring unnecessarily complex hardware. We attempted to find that balance in TRIPS when choosing the architectural constraints on blocks.

In this chapter, we described the compiler's intermediate representation for blocks and the algorithms for forming blocks with respect to the TRIPS ISA. In the next chapter, we describe the phases of the compiler that can violate one of the block constraints and how the compiler handles these illegal blocks.

Chapter 4

Satisfying Block Constraints

The block formation algorithms described in Chapter 3 solve the problem of mapping a sequence of instructions into block form. However, they do not address what to do when a block violates one of the architectural block constraints. In this chapter we describe the algorithms for analyzing block constraints, and how the *block splitter* reforms blocks that violate these constraints. We then show how the combination of these two techniques is integrated into the compiler to support phases that have the potential to produce illegal blocks.

Any phase of the compiler that changes the contents of a block runs the risk of making a legal block illegal. In the TRIPS compiler, we require that once a constraint is satisfied any subsequent phase that modifies a block must guarantee that the previously satisfied constraints remain satisfied. Compiler phases can use the block splitter to provide the guarantee or enforce the constraints themselves. The first phase in the TRIPS back end is code generation which may produce blocks that contain more than 128 instructions or 32 LSIDs. Immediately after code generation, the block splitter satisfies these two constraints. From this point on in the compiler any phase that

changes a block must guarantee that these constraints are satisfied. After the initial block splitting phase the compiler performs hyperblock formation. The hyperblock generator combines blocks together using if-conversion and must not produce blocks with more than 128 instructions or 32 LSIDs. How the hyperblock generator meets these requirements is discussed in Chapter 5. Next, register allocation is performed to satisfy the register constraint–there are four register banks that can each issue eight read and eight write instructions per block. The register allocator inserts spill code into blocks if a register assignment cannot be found that satisfies the register constraint. Since spill code increases the number of instructions in a block, as well as the number of load/store identifiers used, blocks with spills are analyzed by the block splitter and reformed as necessary. After register allocation completes all block constraints are guaranteed to be satisfied. Finally, the compiler generates the stack frame and this phase must guarantee that none of the block constraints are violated upon completion.

4.1 Block Analysis

A block is *legal* when it satisfies all block constraints. The compiler provides an analysis phase to determine the legality of blocks and methods to identify when and what constraints are violated. Blocks are lowered gradually after code generation to their final form and each step of the lowering requires more of the block constraints to be satisfied. As transformations are applied to blocks the compiler must discern whether the transformations are legal with

respect to the block constraints. Once the compiler has fixed a constraint (for example, block size is fixed after hyperblock formation completes), then future transformations are required to enforce the constraint. Therefore, the compiler is structured into phases (Figure 2.2), with each phase satisfying the same or progressively more block constraints. Since blocks change whenever transformations are applied, each phase must re-compute the block constraints and perform block analysis to discern block legality.

Any block can be analyzed by first performing block formation (Chapter 3), analyzing the block, and then returning the block to its pre-analysis form. Returning a block to pre-analysis form is accomplished by removing the register read and write instructions and any instructions inserted for write and store nullification. At a high level, determining if a block is legal involves computing the number of machine instructions in the block (complicated by the fact that there is not always a one-to-one mapping between the compiler's intermediate representation and machine instructions), and computing a summation of the constraints. The block analysis phase in the TRIPS compiler computes the following information for each block from its constituent predicate blocks:

Machine Instructions: A machine instruction is an instruction in the TRIPS ISA. Often the compiler will use pseudo instructions that are later lowered to machine instructions. For example, the TRIPS compiler provides a pseudo instruction for a 64-bit immediate move, which depending on the size of the immediate requires between one and four machine instructions.

In this case, the block analysis phase will examine the size of the immediate to determine how many machine instructions are required. Sometimes, the number of machine instructions cannot be determined because the final form of the instruction is unknown. For example, until register allocation is performed the size of a stack displacement is unknown. In this case, the block analysis phase will analyze instructions conservatively to provide a worst-case bound.

Fanout Instructions: Since there is limited encoding space in instruction formats for target operands, the compiler builds a fanout tree using copy instructions to distribute an instruction's result to its consumer instructions. In the TRIPS compiler, fanout trees are added during instruction scheduling since the optimal tree is often dependent on the schedule. Therefore, the total number of instructions in a block is the sum of the machine instructions plus any missing fanout instructions that would be inserted during scheduling. Block analysis computes the number of fanout instructions using the same algorithms employed by the scheduler when possible or with a conservative but correct approximation when not.

Load/Store Instructions: The compiler computes the number of load and store identifier for each predicate block and the overall highest identifier for the entire block. As an aid to the block splitter during register allocation, the compiler also computes the number of machine instructions needed to represent memory instructions that are used for register spilling. The compiler also notes if a block contains dummy store instructions inserted for store nullification.

Register Usage: We have structured the compiler such that the register constraints are ignored by phases running before register allocation. After register allocation, we compute the total number of register reads and writes in a block, along with the number of slots used in each register bank.

Branch Instructions: The TRIPS ISA has different types of branch instructions. Block analysis provides information on which predicate blocks contain branches since these denote the exits from the block. It also provides higher level information on the types of branches in the block. Specifically we denote when a block contains a branch that is part of a switch statement, has a function call, or is a return from a function call. This information is used by block splitting and hyperblock formation. For example, hyperblock formation cannot merge two blocks if there is a call from one to the other.

4.1.1 Computing Fanout

Determining the number of machine instructions in a block is relatively straightforward once missing instructions have been inserted by block formation. However, computing the fanout in a block is complicated by the fact that fanout instructions are not added until scheduling. TRIPS supports three move instructions: MOV2, MOV3, and MOV4, that can target two, three, or four instructions respectively, with potentially longer latencies and more restrictions on how they can be used. Which instruction the scheduler uses is based on the dataflow graph. For example, the MOV4 instruction requires all targets to reside in certain execution tiles and only during scheduling

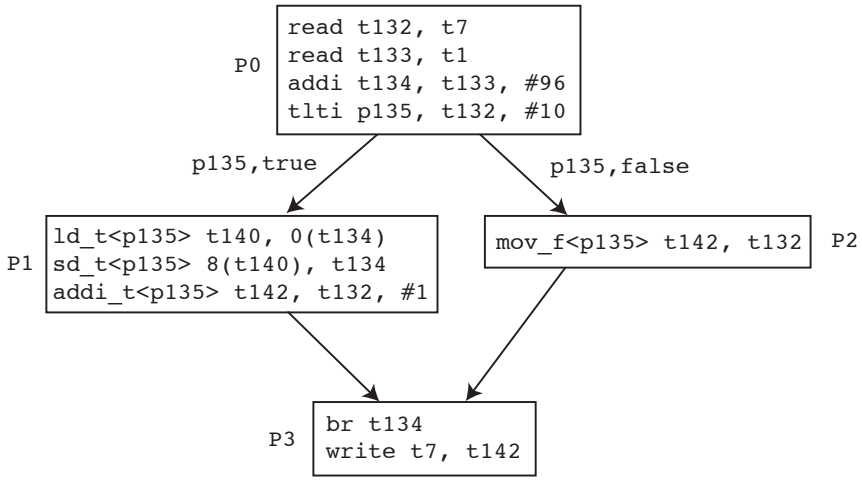


Figure 4.1: Fanout example

can the compiler determine if it can use a MOV4. Therefore, we compute the number of fanout instructions in a block using only MOV2 and MOV3 instructions, potentially reducing block efficiency, but this overestimate is correct and conservative.

MOV2 instructions can target at most two operands. This is the most flexible move instruction since there are no restrictions on the target operand type with the drawback of supporting the fewest targets. The MOV3 instruction can target three operands with the restriction that the operands must all be of the same type. For example, a MOV3 can target the predicate operand of three different instructions but not the predicate operand of one instruction and the left operand of two others. The amount of fanout required depends on the type of move instruction used. If only MOV2s are used then the number

Operand	Targets	Uses	Mov3 Possible
t132	2	3	Yes
t133	2	1	Yes
t134	1	3	No
p135	1	4	Yes
t140	1	1	Yes
t142	1,2	1	Yes,Yes

Table 4.1: Computing the amount of fanout in a block.

of MOV2s required is *uses minus targets*. If a MOV3 instruction can be used, the TRIPS scheduler will use a combination of MOV2 and MOV3 instructions where the MOV3 instructions are at the bottom of a balanced tree.

Table 4.1 summarizes the information necessary to compute the number of fanout instructions for the block in Figure 4.1. This table gives the number of targets supported by each instruction that defines an operand. For example, immediate instructions (`addi`, `tlti`) and load instructions (`ld`) in the TRIPS ISA can target a single consumer [46]. The table also lists the number of uses for an operand. The compiler must insert fanout whenever the number of uses exceeds the number of available targets. The final column denotes whether or not a MOV3 instruction can transfer an operand from its producers to its consumers.

Since we account for the instructions inserted for fanout at predicate block boundaries, we must decide how to associate the fanout with the predicate blocks. One possibility is the fanout for an operand is added to the predicate block containing the instruction that defines the operand. The other

Predicate Block	Machine Instructions	Definition		Use	
		Fanout	Size	Fanout	Size
P0	2	4	6	0	2
P1	3	0	3	2	5
P2	1	0	1	2	3
P3	0	0	0	0	0

Table 4.2: Fanout can be associated with uses or definitions in the predicate flow graph.

option is to account for fanout in the predicate block that contains the consumer instruction.

Table 4.2 shows the fanout and predicate block sizes using the different strategies for Figure 4.1. When the move instructions for fanout are added to the predicate blocks that define the operands, $P0$ appears twice as large compared to $P1$, even though $P1$ has more machine instructions. This leads to poor decisions during analyses based on block size such as block splitting. When the fanout is accounted for in the predicate blocks that contain the consumer instructions, it is more evenly distributed throughout the predicate blocks. Both methods compute the same total block size, however we use the latter since it is balanced. Once block sizes are accurately estimated, a mechanism is still needed to handle blocks that violate the block constraints.

4.2 Block Splitting

Any phase of the compiler that changes the instructions in a block has the potential to violate one of the block constraints. For example, instruction

selection might create a long sequence of code over the 128-instruction limit, or the register allocator might spill, adding additional load and store instructions that exceed the 32 load/store identifier limit. The compiler provides a framework for *block splitting* that can reshape illegal blocks with respect to the block size and load/store identifier constraints into legal blocks. The block splitter reforms a block by splitting it into multiple legal blocks by either *cutting* unpredicated regions of code, or *reverse if-converting* [80] predicated regions.

The TRIPS compiler uses block splitting in two ways: (1) as a pre-pass to hyperblock formation, and (2) during register allocation to split blocks with spill code that have violated one of the block constraints. Block splitting is the dual of hyperblock formation. In hyperblock formation, the compiler tries to identify regions of code to if-convert and place inside the same block. In block splitting, the compiler tries to identify regions of code to remove from a block and place inside another block. Since developing heuristics for both is challenging, we realized we could simplify the compiler by starting hyperblock formation with legal blocks. We also found that it was simpler to make good decisions about where to split basic blocks versus reasoning about complex regions of predicated code. Therefore, we structured the compiler to perform block splitting immediately after code generation as a pre-pass to hyperblock formation, so all blocks being input in to the hyperblock generator are legal. In the case of register allocation, the block splitter has to split only the blocks that the register allocator has spilled in and have a violation. Since spilling in

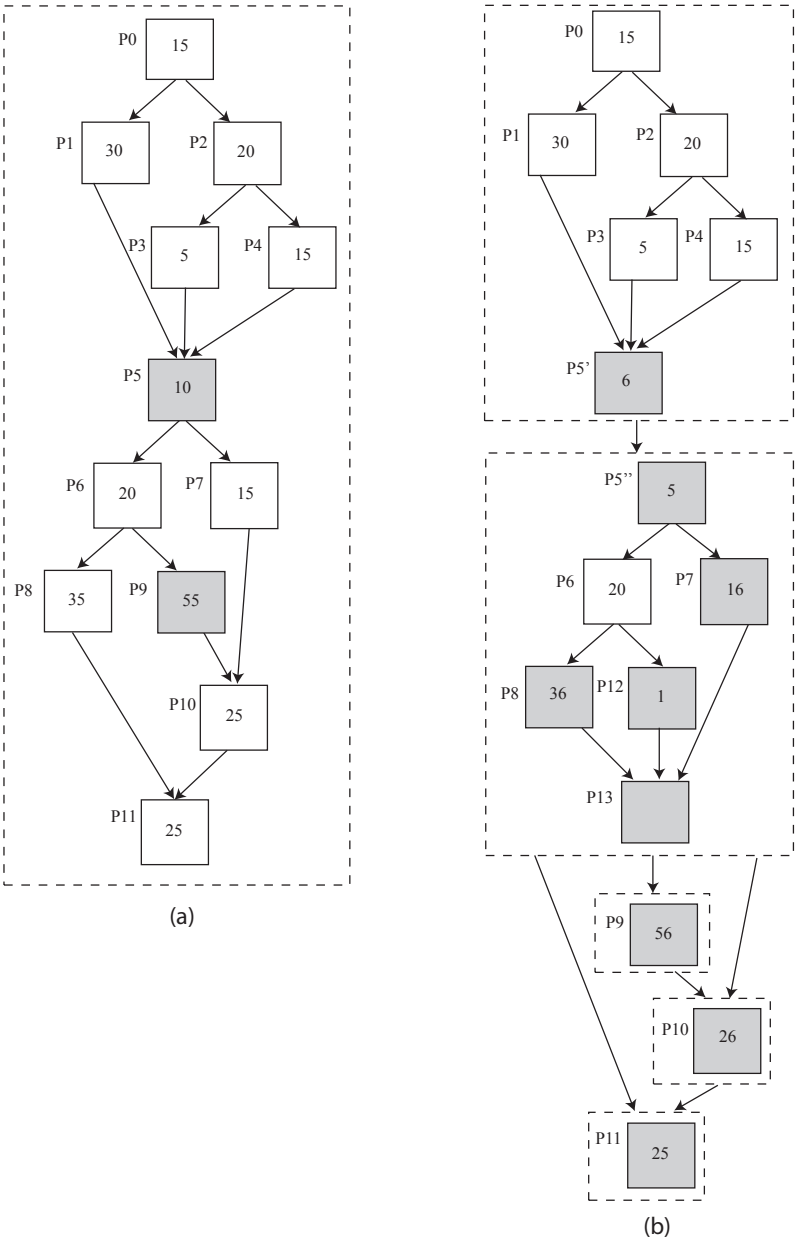


Figure 4.2: Example of a block (a) before and (b) after block splitting. The number of instructions in each predicate block are shown in the nodes. The shading represents predicate blocks that are created or modified by splitting.

the compiler is rare, the fraction of blocks that must be analyzed and reformed is small.

Figure 4.2 shows a block before and after block splitting. The original block contains 260 instructions and must be divided into a minimum of 3 blocks. To form one legal block, predicate block $P5$ is cut into two five-instruction predicate blocks $P5'$ and $P5''$. A branch instruction is added to $P5'$ to $P5''$ increasing the block size for $P5'$ to 6 instructions. Then all of the predicate blocks from the root of the PFG to $P5'$ become one new block. Next, the block splitter chooses to reverse if-convert $P9$ by removing all the predicates from the instructions in the predicate block. Since $P7$ and $P9$ both branch to $P10$, $P10$ cannot reside in the same block as either of these predicate blocks. Therefore, the block splitter must move $P10$ (and by the same reasoning $P11$) into its own block. Branches are added to $P7$ and $P9$ that branch to the new block containing $P10$, to $P8$ and $P10$ to the new block containing $P11$, and a new predicate block $P12$ is created with a branch to the block containing $P9$. Also, an empty unpredicated merge $P13$ is added to maintain the structure of the PFG. The block splitter then checks all the blocks again for legality and iterates if there are any illegal blocks. After block splitting the original illegal block has been split into 5 legal blocks.

4.2.1 Where to Split a Block?

The block splitter must choose where to split a block. In Figure 4.2 when $P9$ was chosen as a split point this choice resulted in $P10$ and $P11$ being

reverse if-converted and moved into new blocks. If $P8$ could be used as the split point instead then the block splitter would not have needed to reverse if-convert $P10$ resulting in four blocks instead of five. Since blocks are padded with NOPs if they do not contain 128 instructions there is a penalty for under filled blocks in the TRIPS prototype. Block splitting should therefore strive to produce the minimum number of splits.[1] A second consideration is whether or not instructions should be packed in blocks or split evenly among blocks. For example, a 300 instruction block can be split into three blocks with 128, 128, and 44 instructions each or split evenly into three 100 instruction blocks. We prefer to split blocks evenly since this leaves some room for spills and optimizations that increase block size and also simplifies scheduling as there are fewer instructions to place.

We split a PFG starting at the root and working breadth first by level. We compute a running total, and continue to the next predicate block only if the current one does not violate the size and LSID constraints. Once we reach a predicate block that causes the block to become illegal we examine all the predicate blocks in the current level and use the following heuristics to determine where to split:

- *Last legal unpredicated predicate block.* Unpredicated code is easier to split than predicated code. Also, unpredicated code does not result in cascading splitting of other predicate blocks. The compiler records the

[1]For TRIPS, minimum # of splits = (block size + 127) / 128

last predicate block found with unpredicated code that was legal when added to the block (i.e. $P5$ in Figure 4.2), and uses it as the location for splitting if the block size including the predicate block is larger than or equal to the average split size.[2]

- *Illegal unpredicated predicate block.* If the current level in the PFG only contains a single unpredicated predicate block, and adding the predicate block to the PFG would cause it to become illegal, the block splitter uses this predicate block as the split point.

- *First illegal predicate block in the level.* It may be possible to include some subset of the predicate blocks in a level of the PFG and have the block remain legal. We examine each predicate block in the level and compute the size of the block and the maximum LSID if the predicate block where included (excluding other predicate blocks in the level). If a predicate block is found that causes the block to become illegal we use it as the split point since we know it cannot possibly be added to the block.

- *Predicate block with multiple successors.* If all the predicate blocks in the level can individually be added to the block we must choose one predicate block as the split point. If we split before a predicate block with multiple successors we can potentially (a) avoid the cascading reverse if-conversion problem, and (b) capture an entire hammock in a single block.

[2]Average split size = total block size / minimum # of splits

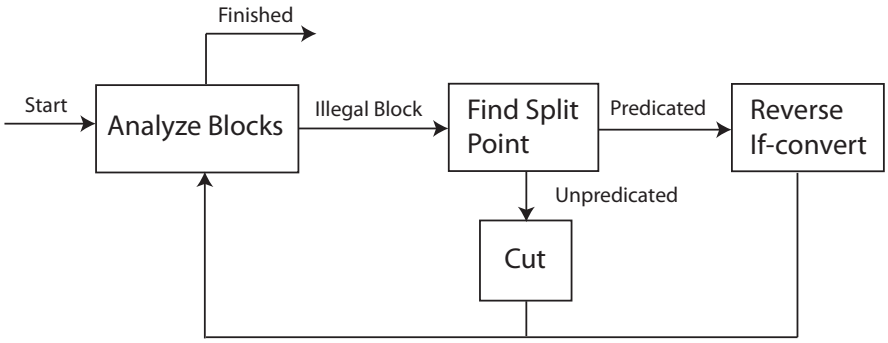


Figure 4.3: Block splitting algorithm.

- *Largest predicate block.* If none of the predicate blocks match the previous heuristics, the block splitter chooses the predicate block with the largest block size.

4.2.2 How to Split a Block?

Block splitting begins by analyzing all the blocks in the BFG. The analysis phase transforms all the blocks into block form (Chapter 3) to determine if a block violates the block size or LSID constraints. If all the blocks are legal, block splitting is complete. Otherwise, for each illegal block, the block splitter finds a split point in the block's PFG using the previously discussed heuristics, and reverse if-converts or cuts the predicate block. The compiler iterates until there are no violations and after block splitting completes, all of the blocks are guaranteed to meet the block size and LSID constraints, but not the register constraints.

Cutting: Block splitting will only cut a predicate block when it is the only predicate block in the PFG. If the block splitter chooses an unpredicated predicate block that occurs in the middle of the PFG (i.e. $P5$ in Figure 4.2) as the split point, it is reverse if-converted into a new block instead of being cut. Predicate blocks are cut based on block size and LSID. If there are more than 32 LSIDs but the block size is legal, the splitter cuts after the instruction with the 32nd LSID in the predicate block. If the LSIDs are legal then the predicate block is cut in half based on the size. If both the LSIDs and block size are illegal the splitter cuts at whichever point is reached first in a forward pass of the instructions. The compiler uses the block analysis from Section 4.1 to compute the size, taking into account the number of real instructions and fanout. Once an instruction is found that can be used as a split point, the block splitter creates a new block and label, and moves the instructions beginning with the split point after the label. Then a branch is added in the original block to the new label.

Reverse If-conversion: Given a predicate block to reverse if-convert, reverse if-conversion creates a new block beginning with the predicate block, removes the predicates from the instructions in the predicate block, adds a label to the beginning of the instructions, and inserts branches to the label in the original block. This process is complicated by the fact that reverse if-conversion can lead to additional predicate blocks being reverse if-converted. In Figure 4.2, $P10$ is on the path of execution for both $P7$ and $P9$. When $P9$ is reverse if-converted, if $P10$ is placed into the same block as $P9$, then $P7$

must some how be able to jump into the middle of the new block, without the instructions in $P9$ executing. Since the compiler does not support hyperblock re-entry [5] this is impossible and $P10$ must also be reverse if-converted. An alternative to reverse if-converting $P10$ is to tail duplicate the instructions into $P7$. However, we leave such decisions up to the hyperblock formation phase which has the mechanism and heuristics to implement them.

To determine which predicate blocks must be reverse if-converted, the compiler performs a depth-first search from the split point adding predicate blocks to be reverse if-converted to a worklist. For each predicate block PB in the search, if a predecessor of PB has not been visited, or if a predecessor of PB is on the worklist, and PB has more than one predecessor, PB is added to the worklist. Next, the predicate blocks on the worklist are ordered based on their depth from the root of the PFG using a breadth first search. If the current predicate block visited by the search is contained in the worklist, it is removed and pushed on a stack. Once done, the predicate blocks are reverse if-converted in the order they are popped off the stack.

4.3 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. Since most operands have a low degree of fanout, with short live ranges [41], this feature of the ISA reduces the number

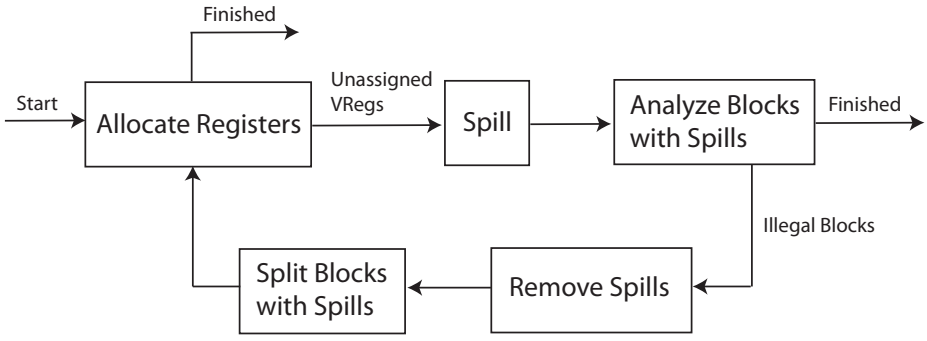


Figure 4.4: Register allocation.

of reads and writes to general-purpose registers by replacing them with direct instruction communication; reservation stations store and forward values in the hardware.

For TRIPS, there are 128 registers divided into four register banks with 32 registers apiece. Each block is limited to eight register reads and eight writes per register bank (for a total of 32 register reads and 32 writes) to simplify the block renaming/forwarding logic. 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. These features require additional state to track and enforce register bank constraints.

The TRIPS compiler uses a modified linear-scan register allocator [22, 57, 59, 68, 75]. Live ranges are computed on a block (BFG node) granularity where a virtual register is live if and only if it is defined and used in different blocks. The allocator gives priority to each virtual register based on its defini-

tions, uses, and spill costs. Subsequently, the allocator assigns virtual registers to physical registers or spills each live range in priority order. Spilling introduces load and store instructions, and thus may cause a block to violate the block size or LSID constraints. The block splitter therefore reexamines every block in which the allocator inserted spill code. If any block is illegal, the compiler removes the spill code from the blocks, splits each block with a violation, and then repeats register allocation. Iterative block splitting is guaranteed to terminate eventually since it strictly reduces the number of instructions in a block. With the large register file and reduced number of live ranges that must be register allocated due to the EDGE block model, the allocator rarely spills. However, more aggressive block formation may expose the need for additional enhancements or a graph coloring allocator that may spill less [75].

4.3.1 Assigning Physical Registers

To determine the assignment order, we compute the strength of each live range, and assign those with higher strengths first. We use an ordered list of available physical registers for each live range and assign caller saved registers before callee saved registers. The allocator selects a physical register from this list and checks if it would cause any constituent 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. When the allocator cannot satisfy the banking constraints in all blocks it must spill.

The order in which virtual registers are allocated to physical registers is important. One possibility is to allocate virtual registers with the shortest live ranges first since spilling them is less likely to be effective. In Figure 4.5, there are three live ranges, $L1$,$L2$, and $L3$ where the length of the bars correspond to the length of the live ranges. A shortest first policy would allocate $L2$, followed by $L3$, and $L1$. Consider what happens when one of the live ranges is spilled. There are five blocks in the figure with the size of the blocks noted at the top. If $L3$ is spilled for example into the 125 instruction block, and the block exceeds the 128 instruction limit, it will be split. This not only affects the quality of the code as blocks are padded with NOPs, but increases the time required for register allocation since splitting forces the allocator to run again. An alternative to the shortest first policy is to order live ranges based on the constraints of the blocks. For example, if the live ranges in the figure are assigned by the size of the blocks they span, the assignment order would be $L3$, $L1$, $L2$. This policy though favors long live ranges. Instead, we compute strength using the length of the live range biased by the size of the blocks spanned by the live range.

$$VR = VirtualRegister$$

$$UD = blocks\,that\,use\,or\,define\,VirtualRegister$$

$$Strength(VR) = 1.0\,/\,length + \sum_{i \in UD} 1.0\,/\,(MaxBlockSize - BlockSize_i)$$

Using this heuristic the strengths of $L1$, $L2$, and $L3$ are 0.30, 0.53, and 0.69 respectively. $L3$ is allocated first, followed by $L2$, and $L1$. Notice that even though $L1$ and $L2$ span a block with 90 instructions, since the block is

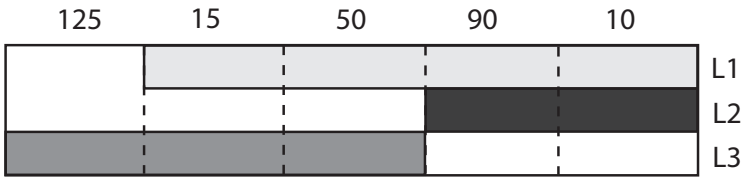


Figure 4.5: An example of three overlapping live ranges. The numbers on the top are the size of the blocks spanned by the live ranges.

not close to the 128 instruction limit, the bias to the strength is small. This is exactly right as we only want to give priority to live ranges that span blocks that are close to the block size limit. We can further extend the heuristic to account for live ranges that span blocks that are close to the LSID limit.[3]

4.4 Stack Frame Generation

Once register allocation is complete the stack frame is generated. Except for the block constraints, generating the stack frame for TRIPS is no different than any other architecture and complete details can be found in the TRIPS ABI [69]. Care must be taken though to emit only legal blocks. As the instructions comprising the stack frame are unconditionally executed, the block constraints can be tracked as the instructions for the stack frame are generated. For TRIPS, the compiler only needs to track the register and LSID constraints since those will be reached before the block size constraint. When

[3]Length includes all blocks that the live range crosses, where as the block constraints only consider blocks that use or define the virtual register.

```
                                        mov SP_caller, SP_callee
                                        addi SP_callee, SP_callee, #256
mov SP_caller, SP_callee                sd 0(SP_callee), SP_caller
addi SP_callee, SP_callee, #256         sd 8(SP_callee), LR
sd 0(SP_callee), SP_caller              sd 108(SP_caller), R12
sd 8(SP_callee), LR                     sd 116(SP_caller), R16
sd 108(SP_caller), R12 ; Bank 0         sd 124(SP_caller), R20
sd 116(SP_caller), R16 ; Bank 0         sd 132(SP_caller), R24
sd 124(SP_caller), R20 ; Bank 0         sd 140(SP_caller), R28
sd 132(SP_caller), R24 ; Bank 0         sd 148(SP_caller), R32
sd 140(SP_caller), R28 ; Bank 0         sd 156(SP_caller), R36
sd 148(SP_caller), R32 ; Bank 0         sd 160(SP_caller), R40
sd 156(SP_caller), R36 ; Bank 0         mov R40, SP_caller
sd 160(SP_caller), R40 ; Bank 0         br masaaki$prlg$1
sd 164(SP_caller), R44 ; Bank 0 Violation

                                        masaaki$prlg$1:
                                        mov SP_caller, R40
                                        sd 164(SP_caller), R44
```

Figure 4.6: Example of a stack frame with a bank violation.

a violation is detected the compiler creates a new block and continues generating the stack frame in this block. The compiler initially places the prologue and epilogue code in two separate blocks from the rest of the function. After the stack frame is generated, the compiler tries to merge the prologue with its succeeding predicate block and the epilogue with its preceding predicate block in the PFG to improve performance [67].

4.4.1 Splitting the Stack Frame

To track the register usage, the compiler models the reads and writes with an array of size equal to the number of banks. Before appending an instruction to the block, the compiler computes the register bank used by the instruction's operands, and uses the bank as an index to the array. If the total number of accesses to the bank is fewer than the total number of supported accesses for a block (in the case of TRIPS eight accesses per bank), the number of bank accesses is incremented in the array, and the instruction is appended to the block. Otherwise, if the number of bank accesses is equal to the number supported by the block, appending the instruction would violate the register block constraint, and the compiler splits the block at the last instruction and appends the violating instruction to the new block. The compiler handles LSIDs at the same time as global registers by tracking the highest LSID in the block. Since the load and store instructions in the stack frame are unconditionally executed, the highest LSID used is equal to the total number of load and store instructions in the block.

When the stack frame is split, the compiler can use the last saved non-volatile register to communicate the updated stack pointer across blocks. A mov instruction is appended to the end of the original block that copies the updated stack pointer to the last saved non-volatile register. Then a second mov is appended to the beginning of the new block that copies the non-volatile register back to the caller's stack pointer.

The example in Figure 4.6 is a typical function prologue where several

non-volatile registers are saved. R12 is stored to the stack and is the first access to bank zero (determined by 12 mod 4 assuming 4 register banks). Next, R16 is stored, followed by R20 and so on. If there is support for eight accesses per bank as in TRIPS, then R40 is the last register that can be saved to the stack. The use of R44 would exceed the number of bank accesses and therefore a new block is created. A branch to the new block is added to the original block along with a copy to save the stack pointer. Then the stack pointer is restored in the new block and R44 is saved.

4.5 Related Work

Compilers for VLIW machines applied reverse if-conversion to balance the amount of speculative execution and to achieve the benefits of a larger window for ILP optimization and scheduling, even in the absence of hardware support for predication [6, 9, 80]. EDGE compilers can also utilize reverse if-conversion for these purposes, but more importantly rely on it to transform illegal blocks into legal ones during block splitting.

Banked register files have been employed in both embedded processors and DSPs [30] to reduce energy requirements and the complexity of bypassing. Generally, a functional unit is associated with a register bank, and non-associated register bank accesses require that the data be moved to the associated bank. This data movement is often compiler managed, complicating register allocation [26, 55]. The TRIPS ISA instead provides uniform access to all banks, but restricts the number of bank accesses per block. 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.

The direct instruction communication used between instructions within blocks, converts live ranges that would otherwise require registers, into intra-block dataflow edges. Since most operands have a low degree of fanout, with short live ranges [41], a block-based execution model uses fewer registers than a conventional architecture. Other architectural models have been proposed which capitalize on this observation. Braids [76] are compiler-formed dataflow graphs similar to blocks. Unlike blocks though, which use predication to aggregate instructions from multiple basic blocks, braids encapsulate subgraphs of basic blocks that are executed within a traditional out-of-order pipeline.

4.6 Summary

Code generation, register allocation, and stack frame generation can all produce illegal blocks that violate one or more of the ISA block constraints. The compiler provides an analysis phase for determining the legality of blocks with respect to the block constraints, and a block splitting framework for reforming illegal blocks into legal ones. Both block analysis and block splitting rely on the block formation algorithms introduced in Chapter 3.

This chapter described the techniques for analyzing and reforming illegal blocks produced by various phases of the compiler. The combination of Chapters 3 and 4 provide enough details to build an EDGE compiler that

produces correct and legal code. In the next chapter, we describe how the compiler optimizes predicated blocks to improve performance.

Chapter 5

Optimizing Predicated Blocks

An EDGE architecture performs best if the compiler minimizes the number of blocks and fills each one with useful instructions. In the absence of predication, a block is simply a basic block. However, to maximize performance, the compiler forms hyperblocks by combining regions of the control flow graph using if-conversion [2]. Hyperblocks provide the compiler with a larger scheduling window and enable additional opportunities for optimization. In this chapter we describe *iterative hyperblock formation*, which incorporates hyperblock formation and optimizations into a unified framework [42]. Then we present several dataflow predication optimizations applied during iterative hyperblock formation to mitigate predication overheads, improve the overall code quality, and enable additional if-conversion [71]. First, predicate fanout reduction removes predicates based on *intra-block* dependence chains. Second, path-sensitive predicate removal removes predicates from instructions that define *inter-block* values. Finally, dead code elimination and dead predicate block elimination are applied to remove useless instructions and predicate blocks.

5.1 Iterative Hyperblock Formation

The work on hyperblocks builds on previous work on compiling predicated hyperblocks for VLIW machines [7, 9, 43, 45, 54]. 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. In TRIPS, hyperblocks differ in two 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 down untaken paths is a non-issue for TRIPS blocks, since blocks can be committed and deallocated as soon as all of their outputs are received. In VLIW machines the constituent instructions within the VLIW instruction must all be independent, and thus the goal of hyperblock formation is to create and combine independent instructions. The constituent instructions in a block, conversely, can be dependent, so the goal of hyperblock formation in an EDGE architecture is to expose many "good" instructions to the window for power-efficient scheduling.

5.1.1 Where to Perform Hyperblock Formation?

Originally, we developed a hyperblock generator that ran before code generation on a high-level target independent form. We allowed the hyperblock generator to form arbitrarily large hyperblocks that were only limited

by the structural constraints of the control flow graph. We found that the generated blocks were often illegal and had to be reformed by the block splitter. The block splitter ended up performing much of the same work required by a backend hyperblock formation phase. Next, we tried to limit the amount of block splitting by estimating the block constraints from the target independent IR. The resulting hyperblocks were still imperfect. When we underestimated the blocks constraints, the blocks were illegal and had to be reformed in the backend by the block splitter. When we overestimated, we found the blocks were underutilized. To address both of these problems, we decided to implement the hyperblock generator as a target dependent phase of the compiler that worked on the actual machine instructions.

5.1.2 Phase Ordering

The next problem we had to solve was a phase ordering problem. When a block is optimized before hyperblock formation, blocks often contain too few instructions for effective optimization. Integer codes are often cited to contain on average five instructions per basic block, and block-based optimizations will certainly be ineffective with such small blocks.. However, when optimizations are applied after hyperblock formation, the optimizations may reduce a block constraint enough that further opportunities for if-conversion are exposed. We would then like to re-run hyperblock formation, but re-running the phase may lead to additional opportunities for optimization. This phase ordering problem is not uncommon in compilers. The solution most often taken is to find

some fixed order for optimization that gives on average the best results across a range of workloads and accept that some opportunity for optimization may be lost. We take a different approach. Since blocks have fixed size, the scope of block based optimization is in fact small and running optimizations multiple times has little impact on over compile time. Therefore, we use an iterative hyperblock generator, that incrementally forms and optimizes blocks until there are no more opportunities for hyperblock formation and optimization.

5.1.3 Merging Blocks

Figure 5.1 shows the algorithm for *iterative hyperblock formation* [42, 68]. To begin, the compiler selects two blocks from the block flow graph (BFG) and merges them together. Next, the compiler optimizes the newly formed block since optimization may change the block's instructions causing an illegal block to become legal. For example, if dead code elimination is applied to a block with 130 instructions, and the number of instructions in the block is reduced to 120 instructions, the hyperblock generator has produced a legal block. On the other hand, optimizations that duplicate instructions such as tail duplication, loop unrolling, or inlining increase the number of instructions in a block, and may cause a legal block to become illegal. After optimization, the block formation algorithms from Chapter 3 are applied again and the block is analyzed as in Chapter 4 to determine if it violates the block size or load-store identifier constraints.[1] Before hyperblock formation begins, we require that all

[1]Register constraints are handled during register allocation.

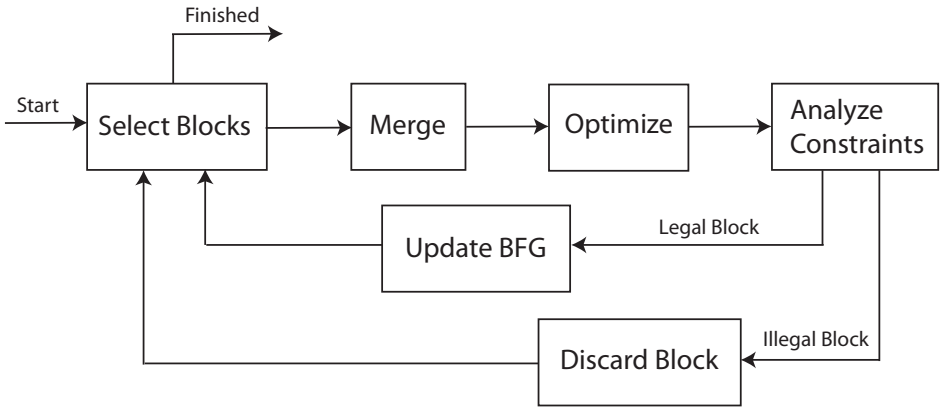


Figure 5.1: Iterative hyperblock formation algorithm.

blocks are architecturally legal with respect to these two constraints (enforced by a block splitting phase preceding hyperblock formation). Therefore, if the analysis determines that a block is illegal, it is a result of hyperblock formation. The block is discarded and the compiler marks the original blocks in the BFG so that it does not attempt to merge them again. If the block is legal, the compiler updates the BFG, replacing the original blocks with the merged block, and the hyperblock generator iterates until complete.[2]

Selecting which blocks to merge is challenging. A good policy will try to fill blocks with many useful instructions while balancing the utilization of compute resources. We explored a variety of policies for selecting blocks during hyperblock formation [42]. One policy that performs well is to select

[2]The hyperblock generator works at the block level, not the predicate block, or instruction level.

blocks using a breadth-first traversal of the BFG, and combine each BFG node greedily with as many of its children as will fit. This allows the compiler to include multiple paths of execution in a block, increasing the amount of instruction level parallelism, and when there is enough room, include entire hammocks in single blocks.

To implement a greedy breadth-first policy, the traversal begins at the root BFG node. When the compiler visits a BFG node it tries to merge the node with a child node unless:

1. The parent node ends with a function call. In the TRIPS architecture, function calls must end inclusion down one control path to avoid jumping into the middle of a block.

2. The child node has other BFG node predecessors. Without tail duplication, the compiler cannot merge children with multiple incoming edges.

After selecting a child block, the compiler makes a copy of the blocks to merge. Merging then proceeds depending on whether or not the parent has multiple exits to the child. If there is a single exit from the parent block to the child (i.e. blocks $B0$ and $B1$ in Figure 5.2), the compiler combines the blocks as follows:

1. The branch instruction leading from the exit predicate block in the parent to the child block is removed, along with the label in the root of the child's PFG.

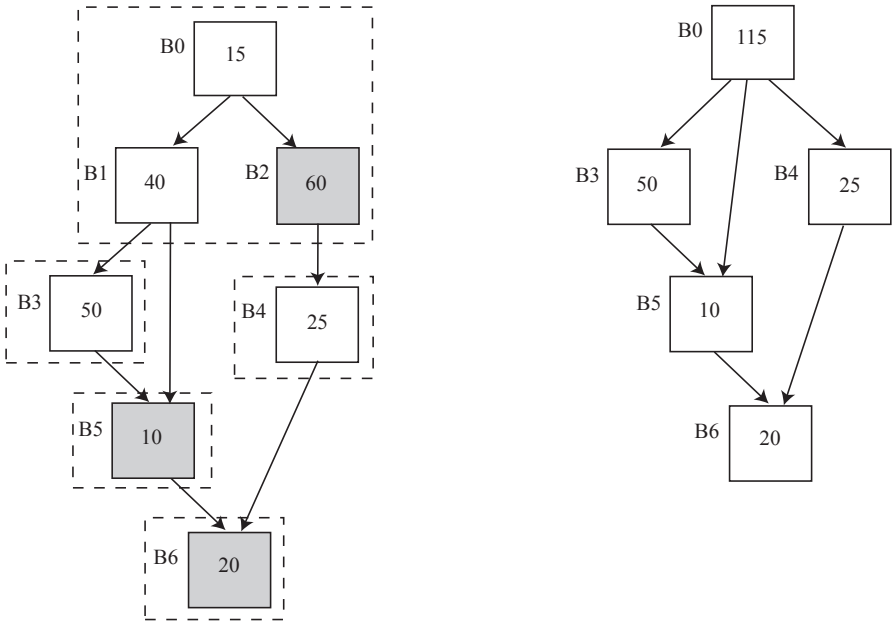


Figure 5.2: A block flow graph before and after hyperblock formation. Blocks to be combined are denoted by dotted lines. $B2$ contains a call that returns to $B4$, while $B5$ and $B6$ have multiple successors that prevent merging.

2. The instructions from the child's root predicate block are then copied into the parent's exit predicate block.

3. If the exit predicate block is predicated, the same predicate is added to the copied instructions.

4. If the root predicate block in the child's PFG has outgoing edges, these become the outgoing edges of the parent's exit predicate block.

However, if there are multiple exits from the parent block to the child block,

the compiler combines the blocks as follows:

1. The branch instruction leading from the exit predicate block in the parent to the child block is removed, along with the label in the root of the child's PFG.

2. If there are exits in the parent block that are not to the child, then a new predicate is created using a move immediate instruction in each of the exit predicate blocks to the child, and the predicate is added to the instructions in the root of the child's PFG.[3]

3. The child's PFG is moved so that it is a successor of all the predicate blocks in the parent with exits to the child.

Figure 5.2 shows a block flow graph before and after iterative hyperblock formation. The compiler starts with the root of the BFG, and attempts to combine $B0$ with $B1$. The combined block $B0'$ is legal and replaces $B0$ and $B1$ in the BFG. $B2$ is then selected for merging with $B0'$ and the combined block $B0''$ is legal and replaces $B0'$ and $B2$ in the BFG. The compiler now chooses $B3$, which is a child of $B0''$, however the merged block is too large (165 instructions) and is discarded. The compiler tries $B0''$ and $B4$ but there is a call instruction in $B0''$ (originally in $B2$) that prevents merging. Since there are no more children of $B0''$ to merge, the compiler moves to the next

[3]A wired-or can also be used but only when predicates have matching conditions: i.e. $\{p135, true\}$ OR $\{p136, true\}$ but not $\{p135, true\}$ OR $\{p136, false\}$.

level in the graph and begins again with $B3$. $B3$ has a single child $B5$, but since there are multiple incoming edges to $B5$, the compiler cannot merge the blocks. There are no more children of $B3$ to try and the compiler moves on to $B4$. Merging $B4$ and $B6$ fails because of the multiple incoming edges to $B6$ and the level is done. $B5$ is then selected in the next level and fails to be merged with $B6$. $B6$ has no children and hyperblock formation is complete.

5.2 Predicate Fanout Reduction

A dataflow predicating compiler can apply both implicit predication and speculative hoisting to reduce predicate fanout, thus eliminating unnecessary predication and avoiding the insertion of move instructions that would otherwise be required to forward predicates to their consumers [71]. For example, in Figure 5.3(a), `tgti` is an immediate test instruction that defines predicate `t3`. If the compiler does not apply predicate fanout reduction then every instruction in predicate blocks $\{t3, true\}$ and $\{t3, false\}$ must be predicated, resulting in seven uses of $t3$. In the TRIPS ISA, immediate instructions only support a single target, requiring the compiler to insert four move instructions (using MOV2s) to fanout the predicate to its consumers.

The compiler performs predicate fanout reduction by either predicating instructions at the top of dependence chains (Figure 5.3(a)), or by predicating instructions at the bottom of dependence chains (Figure 5.3(b)). Predicating at the top, forces all predicates to resolve before any predicated instructions can execute. This strategy results in the longest critical path, as every pred-

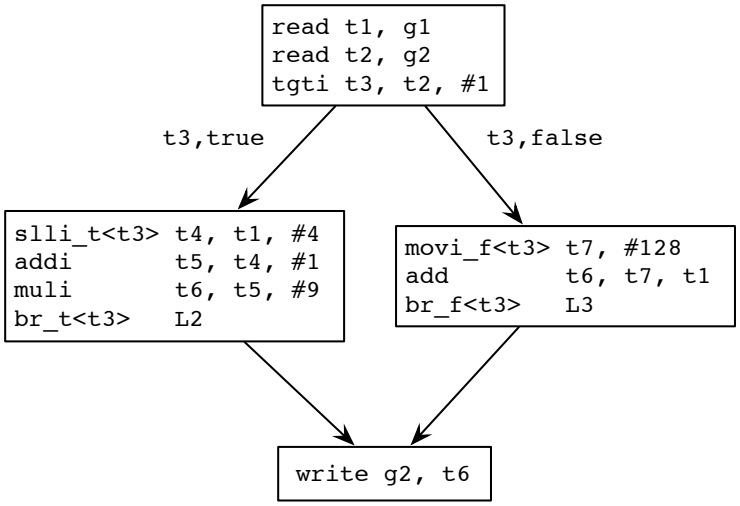


Figure 5.3: A block after predicate fanout reduction. In (a) the top of each dependence chain is guarded by a predicate, while in (b) the bottom of each dependence chain is guarded by a predicate.

icate must fully resolve before execution down a dependence chain can continue. The lack of speculation though may lead to reduced energy utilization. Predicating at the bottom, allows the compiler to shorten the critical path, and increase the amount of speculative execution within a block, which may lead to higher performance. For both cases, predicate fanout reduction is applied using the PFG in static single assignment form. The compiler is free to apply implicit predication and/or hoisting to remove a predicate from any instruction, including instructions that may raise exceptions (subject to the restrictions discussed in Chapter 1).

5.2.1 Predicating the Top of Dependence Chains

To predicate the top of a dependence chain, the compiler examines each predicate block in the PFG. If the predicate block is unpredicated, the compiler continues to the next predicate block. Otherwise, the compiler identifies the instructions that are *not* at the top of a dependence chain as these are the instructions that can be unpredicated.

To identify candidate instructions, the compiler examines every non-phi instruction in a predicate block. For each candidate, the compiler uses use-def information to retrieve the instructions that define a candidate's source operands. The candidate is added to a work list to be unpredicated when the following conditions are all met by any one of the source operands' defining instructions: (1) the defining instruction is predicated, (2) does not define the predicate for the candidate instruction, and (3) is in the same predicate

block as the candidate. Once all the instructions in a block are examined, the compiler removes the predicates from the instructions in the work list.

Figure 5.3(a) shows a block after predicate fanout reduction, guarding the top of dependence chains. The predicates have been removed from the `addi, muli,` and `add` instructions. The predicates must be left on the `slli`, `movi` and branch instructions because they represent the tops of the dependence chains in the block.

5.2.2 Predicating the Bottom of Dependence Chains

To predicate the bottom of a dependence chain, the compiler removes a predicate from an instruction if all of the following conditions are met:

1. The instruction is not a branch or store. Removing the predicate from a branch will lead to incorrect execution when the branch should not have been taken. Likewise, removing the predicate from a store will lead to undefined behavior as the likely outcome is that two stores with the same load-store identifier will execute.

2. The instruction does not define a predicate. Speculative execution of a predicate will trigger speculative execution of all the instructions that are predicated on this predicate. Instructions on the non-speculative path may target instructions on the speculative path leading to some instructions receiving operands twice, which has undefined behavior.

3. The instruction does not define an operand of a SSA `phi` instruction. Phi

instructions represent locations in the block where multiple values merge and are selected from. If an instruction that defines a phi's operand executes speculatively, the `phi` will forward the speculative value to its consumers. When the non-speculative value finally reaches the `phi`, the `phi` and its dependent instructions will need to re-execute. There is no support for such an execution model in the TRIPS processor.

Figure 5.3(b) shows a block after predicate fanout reduction, guarding the bottom of dependence chains. The predicates used by the `slli addi`, and `movi` instructions have been removed. The predicates must be left on the `muli`, `add` and branch instructions because they produce block outputs.

5.2.3 Speculative Loads

Predicating the bottoms of dependence chains allows instructions without external side effects to execute speculatively. However, the compiler must be careful when applying this optimization to load instructions because the TRIPS ISA requires every load-store identifier produced from a block to be unique. Two load instructions that share an LSID, and both execute, will result in undefined runtime behavior. The compiler must guarantee that a load does not share an LSID with another load, before allowing the load to execute speculatively, or use an LSID assignment algorithm (such as the ones presented in Section 3.7) that assigns every load a unique identifier.

If a load cannot be unpredicated, for example, the load defines the operand of a phi instruction, we would still like to speculatively execute the

load. The compiler can insert a conditional move instruction after any load that cannot be unpredicated, and then remove the predicate from the load.

5.3 Path-Sensitive Predicate Removal

The TRIPS ISA requires that all paths through a block produce the same set of register writes. If an instruction defines a register that is live-out from a block on one path but not another, the compiler must insert additional instructions to produce a definition for the register on all paths. The compiler can either read in the register that is live-out and copy this value on the paths without a definition, or insert `null` instructions to nullify the write on these paths.

For example, in Figure 5.3(a), `g1` and `g2` are both live-out. The compiler therefore inserts three additional `mov` instructions to write the original values of `g1` and `g2` back on the paths without the definitions. Two move instructions in the $\{t3, true\}$ and $\{t7, true\}$ predicate blocks set the temporary register `t6` for `g2`. One move instruction in the $\{t3, false\}$ predicate block sets the temporary register `t5` for `g1`. However, the compiler does not need to preserve these registers on paths where they are not live-out.

Path-sensitive predicate removal is an optimization that promotes instructions that define values live-out of a block to execute unconditionally. This optimization reduces the amount of predicate fanout, decreases the overhead for satisfying the constant output constraint for writes (either by reducing the number of null instructions or the register pressure for reading in the

register), and increases speculation through early resolution of inter-block dependences. Potentially drawbacks though are the increased energy utilization and contention from additional speculation when applied to instructions off the critical path.

An instruction is a candidate for this optimization when the following conditions are met:

1. The instruction defines a register that is live-out from a block and the register is not live-out on every path. There is some path of execution in which the value in the register does not matter since the register is not read by the successor blocks.

2. The instruction dominates the exits from the block in which the register is live-out. If the instruction does not dominate all the exits that produce an actual value, then there is some other instruction in the block that writes to the same register. Allowing any of these instructions to unconditionally execute will most likely result in multiple instructions producing a value for the same register write.

3. The instruction cannot raise an exception. Since the value produced by the instruction will be written unconditionally to the register file regardless of the path of execution, there is no way to gate an exception on the paths for which the register is unused. Allowing instructions to produce exceptions (such as divide) on speculative paths violates the exception model (Section 1.3).

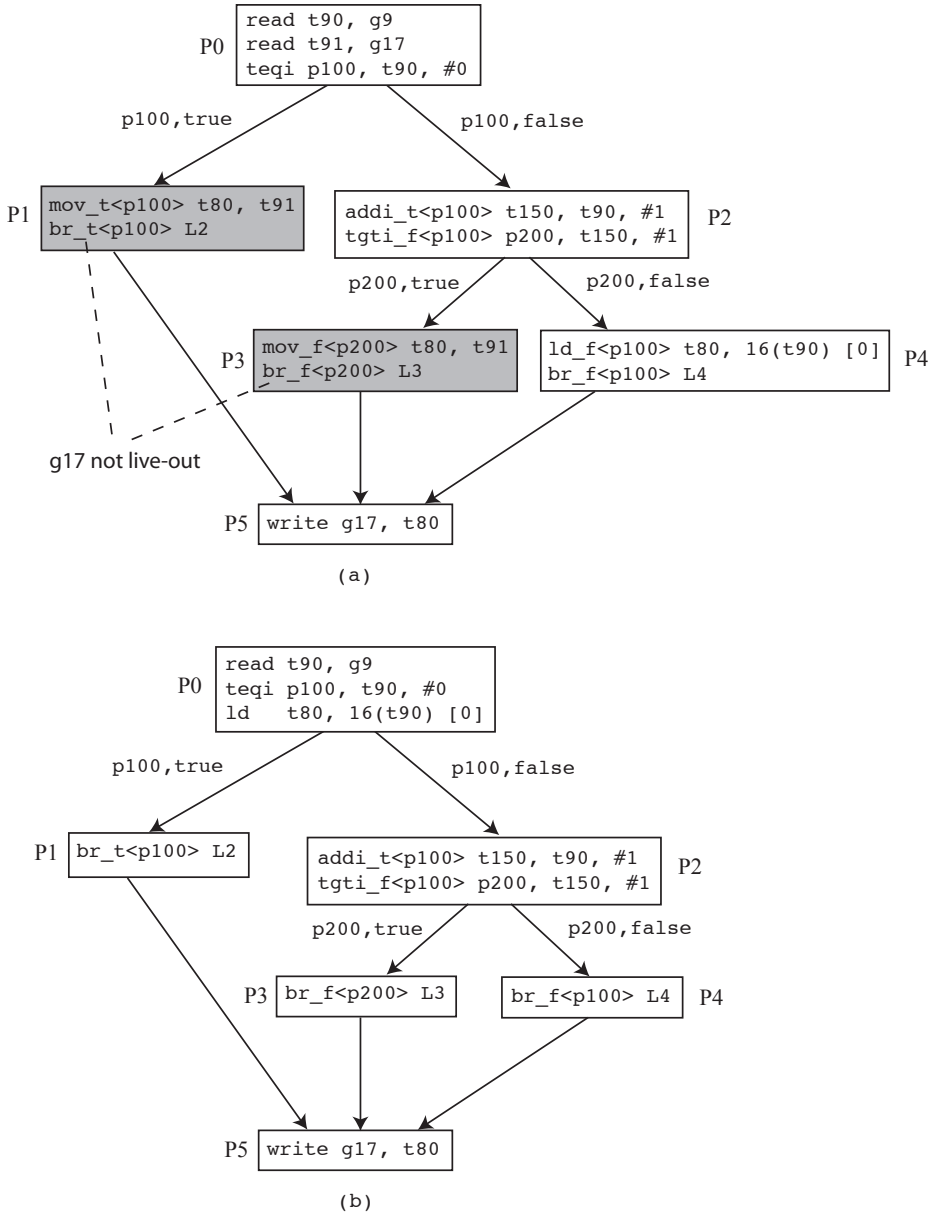


Figure 5.4: In (a) `g17` is only live-out from the exit in $P4$. After applying path-sensitive predicate removal in (b), the load is promoted to execute unconditionally allowing one `read` and two `mov` instructions to be removed.

Any candidate instruction found may be promoted to execute unconditionally, and implies that the instructions that define the candidate's operands, excluding any instructions that define predicates, must also be promoted. The instructions that are part of this recursive promotion must follow the three rules outlined above.

In Figure 5.4(a), `g17` is not live-out from the exits in $P1$ and $P3$. The compiler still has to write some value for `g17` though on these paths, and in the example reads in the original value of `g17` and inserts two conditional moves to forward the value to the write. Since the value that is in register `g17` is only live when the exit from $P4$ is taken, the compiler applies path-sensitive predicate removal to write the same value on all paths as shown in Figure 5.4(b). The `ld` instruction is promoted to the dominating predicate block $P0$, causing it to be executed unconditionally, and enabling the compiler to remove both move instructions in $P1$ and $P3$ along with the register read in $P0$.

5.4 Dead Code Elimination

After optimizations are applied to blocks, instructions can become dead, such that an instruction is never executed or the result of the computation is unused. Dead code elimination removes these useless instructions from a block. The compiler performs dead code elimination in SSA form using the standard mark and sweep algorithm [49]. However, one change is needed to support the dataflow predication model of blocks.

The block in Figure 5.5 contains a phi instruction in $P3$. When trans-

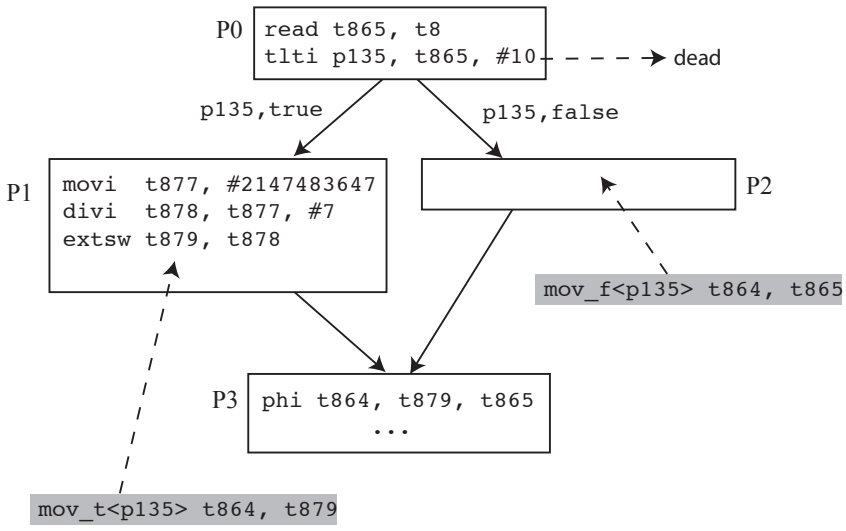


Figure 5.5: Example showing how dead code elimination can incorrectly remove the test instruction which defines a predicate (`p135`), resulting in uses of undefined temporaries after SSA phi removal.

forming out of SSA form back to normal form, the phi instruction will be removed and mov instructions will be inserted in $P1$ and $P2$. The move instructions will be predicated on `p135` which is created by the test instruction in $P0$. Before leaving SSA form, no instructions in the block use `p135`. If dead code elimination is run on this block, the test instruction that defines `p135` will appear dead and be removed. Then during phi removal, the mov instructions will use a predicate that is undefined.

To solve this problem, when the compiler marks useful instructions during dead code elimination, if any useful instruction is a phi, the instructions

that define the predicates for the phi's predecessor predicate blocks are also marked as useful. For example, in Figure 5.5, when the compiler marks the phi as useful, the test instruction that defines the predicate for $P1$ and $P2$ is also marked as useful.

5.5 Dead Predicate Block Elimination

When dead code elimination is able to remove the instruction that defines a predicate, the predicate blocks that utilize this predicate also become dead. Dead predicate block elimination is an optimization that identifies and removes dead predicate blocks from the predicate flow graph. The optimization is run after dead code elimination, whenever dead code elimination removes an instruction that defines a predicate.

To identify dead predicate blocks, the compiler traverse the PFG, and adds any empty predicate blocks that are also predicated to a work list. Then for each predicate block in the work list, the compiler uses use-def information to retrieve the instruction that defines the predicate for the predicate block. If the defining instruction is null, the predicate block is dead and the compiler removes the predicate block from the PFG.

In Figure 5.6, predicate blocks $P3$ and $P4$ are dead. The compiler searches the PFG and adds $P2$, $P3$, and $P4$ to the work list because they are all empty and predicated. Next, $P2$ is examined and the instruction that defines `p100` is retrieved and found to be non-null, signifying $P2$ is not dead. Next, $P3$ is checked and the instruction that defines its predicate is null so $P3$

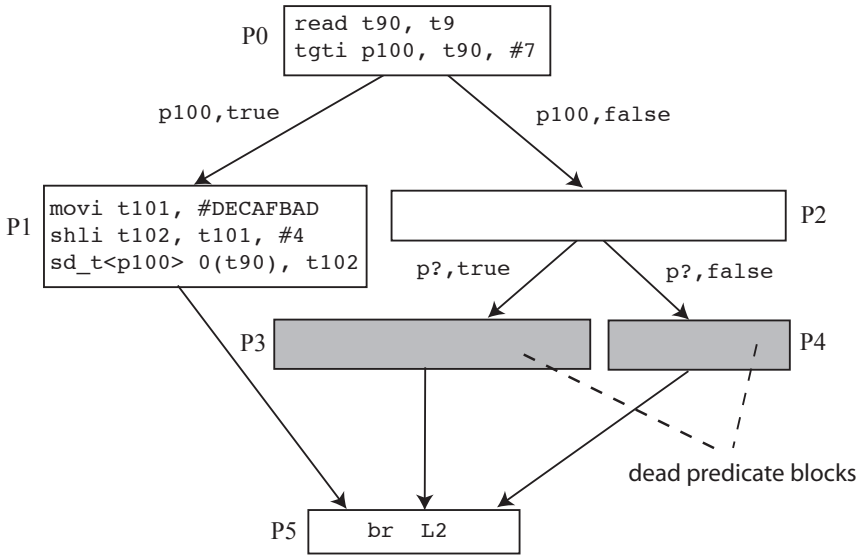


Figure 5.6: After dead code elimination is run, predicate blocks $P2$, $P3$, and $P4$ are empty. Dead predicate block elimination will remove $P3$ and $P4$, but $P2$ is required to maintain the predicate flow graph.

is dead and removed from the PFG. Finally, $P4$ is checked and found to be dead and removed.

5.6 Related Work

The work on hyperblocks builds on previous work on compiling predicated hyperblocks for VLIW machines [7, 9, 43, 45, 54]. VLIW architectures build hyperblocks to maximize exposure of independent instructions for long-word packing. When forming hyperblocks, VLIW compilers scrutinize depen-

dence height in less frequently accessed basic blocks, since that height puts a lower bound on the VLIW schedule. In TRIPS, hyperblocks differ in two 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 down untaken paths is a non-issue for TRIPS blocks, since blocks can be committed and deallocated as soon as all of their outputs are received.

Architectures have used predication since the 1970s. The CRAY-1 implemented predication in the form of vector masks [61] to guard individual vector operations. Predicated execution became more prevalent in VLIW machines in the 1980s and 1990s. The Multiflow Trace machines supported partial predication using the select instruction [40]. The Cydra 5 [58] and the IA-64 Intel Itanium processors' ISAs include a predicate operand with every instruction. Several RISC architectures also support some predicated execution; the in-order ARM processor predicates most instructions, but the out-of-order Alpha and SPARC V9 architectures limit predication to conditional move instructions.

Predication research has generally fallen into two categories: ISA and microarchitecture support for efficient execution, and compiler algorithms and optimizations to use and exploit predication. Allen et al. first described if-conversion to convert control dependences to data dependences [2]. Mahlke et al. proposed the use of hyperblocks as an effective compiler structure for performing predication and exposing scheduling regions to the compiler [45].

Researchers have also shown that predication is effective for enabling software pipelining on loops with control structures [24, 79].

Several solutions have been proposed to alleviate the overheads of predication in VLIW architectures and to a limited extent, in dynamic superscalar architectures. For VLIWs, August et al. propose a framework that mitigates fetch and execution overhead by balancing control speculation and predication [9]. In out-of-order processors, researchers have proposed *predicate prediction*, which predicts the resolution of the predicate in the dispatch logic [18], *wish branches*, which enable the hardware to dynamically and selectively employ predicated execution [37], and *predicate slip*, which delays the use of the guard predicate until commit [78].

Conventional architectures typically save the results of instructions that define predicates in either the general purpose register space or in a private predicate namespace. In addition to specifying two (or more) data source operands, a predicated instruction must also specify its predicate operand. In IA-64, the predicate operand consumes six bits of each instruction. Due to this encoding pressure, some architectures use predication only in a small number of instructions (ARM is a notable exception). For example, Alpha and SPARC V9 architectures offer a single conditional move operation for use in simple control constructs. To extend predication to other instructions, Pnevmatikatos et al. propose using the GUARD instruction [56]. This instruction, in conjunction with a predicate register file, specifies which instruction to guard among the set of successor instructions.

To generate predicates for instructions inside complex control structures, the compiler must invert and merge predicates generated along each if-converted branch. A long predicate computation chain, in addition to increasing instruction overhead, may end up on the program critical path. Researchers have addressed this problem in different ways: by generating complementary predicates [32], by using *wired operators* [32], and by *program decision logic minimization* [8].

Of the conventional architectures, VLIW architectures benefit from low-overhead predication, but lose performance because falsely predicated instructions can lengthen the critical path of execution. Superscalar processors have not benefited from predication due to the complexity of its implementation in an out-of-order microarchitecture. Less conventional architectures, such as historical dataflow machines, have combined only partial predication with dynamic scheduling.

Dataflow predication, as instantiated in the TRIPS architecture, differs from previous partially predicated dataflow architectures in three major ways [4, 25, 74]: First, predicates may directly guard individual instructions, avoiding the need for gate or switch instructions. Second, any instruction can generate a predicate merely by targeting the predicate operand of another instruction. Third, instructions may receive multiple predicate operands before firing. These three features enable dense encoding, as each 32-bit instruction requires only two bits to specify whether it is predicated. They also enable efficient compound predicate computation, since dataflow predication supports

the disjunction of an arbitrary number of predicates, and since predicate-producing instructions may themselves be predicated. Finally, they support implicit predication, since only the input instructions to a dataflow graph need to be predicated to implicitly predicate the entire graph.

Dataflow predication allows the compiler to increase the amount of speculation within and between blocks. Speculative loads are one such optimization that have been successfully employed in both VLIW and RISC architectures to hide memory latency [19, 51]. The previous techniques for load speculation fall into two categories: those that are architecturally hidden (dependence prediction [50]) and those that are exposed by the ISA to the compiler through additional load instructions [19, 31, 48]. The work on compiler support for speculative loads has focused on moving loads above conditional branches (control speculation) and store instructions (data speculation). These techniques rely on the ability to ignore or discard the effects of a misspeculation. Rogers and Li propose the use of poison bits [60], while work on predicated VLIWs have utilized special compiler inserted check instructions [31].

One drawback of speculative loads in previous models is the increased register pressure due to the loading of speculative values. Another problem is the architectural and/or compiler support required to support misspeculation. In an EDGE architecture like TRIPS that supports a dataflow predication model [71], neither of these issues arise. Instructions communicate directly instead of through registers, while the dataflow predication model allows for

misspeculations to be filtered out by taking advantage of the existing predication support in TRIPS.

5.7 Summary

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. To alleviate the tension between optimization and producing legal blocks, this chapter introduces iterative hyperblock formation, which incrementally forms and optimizes blocks. During iterative hyperblock formation, the compiler applies both scalar and predicate optimizations to improve the code quality of blocks, which in turn provides further opportunities for if-converting and merging blocks.

Because EDGE architectures employ direct producer/consumer bypassing instead of automatically broadcasting instruction results through a common register file, delivering a single predicate to many predicated instructions may incur significant overhead. For example, if a basic block is predicated on some predicate p, a naive implementation predicates every instruction in that basic block on p. Due to instruction size limitations, instructions that generate predicates have only one or two targets. Consequently, a naive compiler would build a software fanout tree that distributes the predicate to all the instructions, increasing dependence height and adding overhead to the block.

A dataflow predicating compiler can eliminate most of these predicates by applying the predicate fanout reduction techniques described in this chapter.

In the next chapter, we evaluate the the complexity of the compiler flow developed in this dissertation, and the quality of the code produced using the optimizations from this chapter. We also perform an evaluation of the some of the design decisions made in the TRIPS ISA.

Chapter 6

Evaluation

This chapter evaluates the compiler using experimental results from the TRIPS prototype processor and tsim-arch, the TRIPS ISA functional simulator. We use 21 of the 26 SPEC2000 benchmarks (the five remaining benchmarks are FORTRAN90 and C++ which Scale does not support). We break the evaluation into three parts. First, we present the compile time for the SPEC2000 benchmarks to evaluate the complexity of the compiler flow developed in this research. Next, we evaluate the block constraints imposed by the TRIPS ISA to determine if they are reasonable choices and identify areas for improvement. To reduce the execution time for these experiments we use the MinneSPEC inputs [38]. Finally, we evaluate the performance of the optimizations presented in Chapter 5 using the SPEC2000 reference inputs.

The TRIPS prototype processor is a 170M transistor, 130nm ASIC chip, with two processor cores both running at 366MHz. Each core contains a 64KB L1 instruction cache, 32KB L1 data cache, and is capable of sustaining 16 instructions per cycle. Table 6.1 lists more details off the processor parameters. Previous work [28] places the performance of the TRIPS prototype, measured in cycles, between an Intel Pentium 4 and Intel Core2 processor.

Processor Parameter	Configuration
L1 Instruction Cache	Four 16KB banks, 2-way set associate, 1 port per bank
L1 Data Cache	Four 8KB banks, 2-way set associate, 1 port per bank
Registers	4 register banks, 32 registers per bank, 1 port per bank
Instruction Fetch	16 instructions per cycle
Instruction Issue	16 instructions per cycle
Instruction Commit	16 instructions per cycle
Load and Store Ports	4 effective load and store ports
Control Flow Predication	Predictor using exit histories to predicate the next block, employing a tournament local/gshare predictor similar to the Alpha 21264 with 9K, 16K, and 12K bits in the local, global, and tournament exit predictors, respectively
L2 Cache	1 MB L2 cache, with 5 on-chip network ports to access the L2 cache banks

Table 6.1: TRIPS Processor Parameters

Table 6.2 lists the compiler optimizations used for the evaluation. Optimization level -O3 uses basic blocks as the basis for the architectural blocks, and enables all high-level transformations in the compiler including inlining, loop unrolling and scalar optimizations. Optimization level -O4 uses hyperblocks in place of basic blocks and includes all the optimizations from -O3 as well as back end optimizations that utilize the dataflow predication model. Unless otherwise noted, all results presented in this chapter are for hyperblocks compiled with full optimization (-O4).

Optimization	**Level**	**-O3**	**-O4**
Inlining (10% bloat)	AST	X	X
Loop Unrolling	HIR	X	X
Dead Variable Elimination	HIR	X	X
Basic Block Load and Store Elimination	HIR	X	X
Expression Tree Height Reduction	HIR	X	X
Global Value Numbering	HIR	X	X
Copy Propagation	HIR	X	X
Array Access Strength Reduction	HIR	X	X
Sparse Conditional Constant Propagation	HIR	X	X
Global Variable Replacement	HIR	X	X
Loop Invariant Code Motion	HIR	X	X
Structure Fields In Registers	HIR	X	X
Useless Copy Removal	HIR	X	X
Loop Test at End	HIR	X	X
Hyperblock Formation	Backend		X
Predicate Fanout Reduction (Bottom)	Backend		X
Speculative Loads	Backend		X
Dead Code Elimination	Backend	X	X
Dead Predicate Block Elimination	Backend	X	X

Table 6.2: Compiler optimizations are performed on the abstract syntax tree (AST), high-level target independent IR (HIR), and on blocks (backend). The two optimization levels are -O3 (basic blocks) and -O4 (hyperblocks).

6.1 Compile Time

We measure compile time for the SPEC2000 benchmarks on Linux using an unloaded Dell Workstation with a 3GHz Intel Xeon processor, 1MB of L2 cache, and 2GB of main memory. The Java runtime used is Java SE 6 from Sun Microsystems (version 1.6.0_11).

Table 6.3 gives the breakdown of compile time for each of the benchmarks compiled at -O4. The work in this thesis is measured by the column labeled "backend" and accounts for 3-17% of the overall compile time. The scheduler is measured separately and accounts for between 71-95% of the overall compile time.

The percentage of time spent in the individual back end phases is similar across benchmarks. Therefore, in Figure 6.1 we only show the breakdown of the average time spent in the back end, excluding the time spent scheduling. We divide the time into the following categories: code generation, building the block flow graph and predicate flow graphs, performing block splitting before hyperblock formation, hyperblock formation and optimization, register allocation and stack frame generation, and other.

Code generation accounts for approximately 3.7% of the time spent in the back end. Since our code generator is hand written, reducing this time is unlikely, and this phase makes a good comparison point for the complexity of other phases.

Immediately following code generation the compiler builds the block

Benchmark	**Total (sec.)**	**Frontend (%)**	**Optimizer (%)**	**Backend (%)**	**Scheduler (%)**
ammp	201	2.6	2.0	4.6	89.4
applu	86	1.0	9.7	5.0	82.2
apsi	231	0.7	5.6	5.9	84.6
art	40	3.6	3.1	6.0	84.8
bzip2	71	2.1	4.9	6.0	84.8
crafty	276	2.8	5.8	11.8	74.8
equake	30	3.4	12.8	7.1	73.0
gap	1821	0.6	0.8	3.2	94.5
gcc	1656	1.9	4.7	16.6	71.4
gzip	102	2.9	2.0	4.5	88.7
mcf	38	7.3	2.0	3.7	84.7
mesa	1127	1.9	2.6	9.5	83.8
mgrid	31	1.6	13.1	5.7	76.7
parser	216	1.8	1.5	5.2	89.7
perl	754	2.8	4.3	13.1	74.2
sixtrack	1531	0.9	10.1	12.6	71.5
swim	17	2.6	6.6	5.5	80.8
twolf	804	1.2	1.2	3.4	93.3
vortex	366	4.9	1.5	6.2	85.4
vpr	187	2.3	2.1	5.0	88.7
wupwise	53	3.1	2.5	4.7	86.0

Table 6.3: Breakdown of compile time with full optimization (-O4). The time spent in the back end does not include the scheduler which is shown separately.

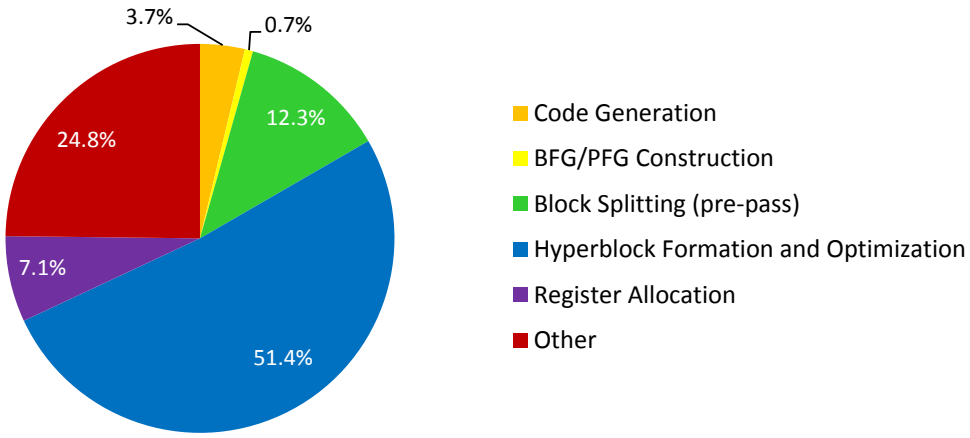


Figure 6.1: Breakdown of the average back end compile time (excluding scheduling).

flow graph and predicate flow graphs, requiring 0.7% of the back end time. Both of these algorithms are light-weight as they examine each instruction only once to build the graphs.

Next, the block splitter runs to prepare for hyperblock formation. This phase is a good measure of the complexity of the block formation algorithms from Chapter 3 since block splitting must analyze each block, which requires entering block form. On average 12.3% of the time in the back end is spent in block splitting.

The hyperblock formation and optimization phase accounts for the largest percent of time at 51.4%. This large percentage makes sense since the block formation algorithms and predicate optimizations are applied every time two blocks are merged. Still the average overall compile time spent in

this phase is only 25 seconds.

Register allocation and stack frame generation accounts for 7.1% of the back end time. Since most benchmarks do not have spills the block splitting and analysis phases never run. When they do run the number of blocks with spills that are illegal and require splitting is low. However, splitting does force the register allocator to run again which increases compile time. If we used a graph coloring allocator, any spill would result in the allocator running again. Therefore, the re-allocation time is a slight drawback when using linear scanner but would be required anyway with graph coloring.

The remainder of the time is shown as "other" and represents the time spent applying peephole optimizations, expanding pseudo instructions into machines instructions, and writing the TIL file to disk.

6.2 Block Constraints

In this section, we evaluate the four block constraints imposed by the TRIPS EDGE ISA on the compiler: maximum of 128 compute instructions per block, 32 load-store identifiers per block, eight register reads and eight registers writes to each of four banks per block, and constant output (each block must always generate a constant number of register writes and stores, plus exactly one branch). We use the TRIPS prototype processor to measure cycle counts for each benchmark, and the TRIPS function-level simulator tsim-arch to produce a block profile that captures the number of blocks and the dynamic number of non-speculative instructions that execute.

6.2.1 Block Size

Table 6.4 shows the average number of instructions per block when using basic blocks and hyperblocks. These numbers are for committed instructions and do not include speculative instructions with non-matching predicates. Basic blocks have on average 14 dynamic instructions per block while hyperblocks double the dynamic block size to 30. With hyperblocks, parser has the smallest average dynamic block size of 14, and effectively utilizes only 11% of the instruction window. Mgrid has the largest dynamic block size with 44 instructions and utilizes 34% of the available instruction window. To continue to increase block sizes, the compiler must apply optimizations that remove structural constraints and limit hyperblock formation (such as performing tail duplication to remove merges in the block flow graph, or peeling loops in the backend to provide additional instructions to merge).

To determine if the 128 instruction limit was an appropriate choice for the block size of the TRIPS prototype, we compiled to an hypothetical ISA with support for an unlimited number of instructions per block and measured the size of each block immediately after hyperblock formation. Then we sorted the blocks by power-of-two block sizes and computed the total number of blocks in each bin. For example, if a block had 300 instructions, the block would be added to the bin for 512 instruction blocks.

Figure 6.2 shows the distribution of block sizes for each benchmark. The largest block size required was 1024 instructions, while the smallest block size required was a single instruction. Using 128 instruction blocks captures

Benchmark	Dynamic Block Size		Increase
	Basic Blocks	Hyperblocks	Factor
ammp	5	24	4.8
applu	27	48	1.8
apsi	39	49	1.3
art	10	43	4.3
bzip2	11	25	2.3
crafty	10	29	2.9
equake	11	26	2.4
gap	9	21	2.3
gcc	8	20	2.5
gzip	10	36	3.6
mcf	7	32	4.6
mesa	15	25	1.7
mgrid	44	44	1.0
parser	5	14	2.8
perl	10	17	1.7
sixtrack	14	28	2.0
swim	23	41	1.8
twolf	9	28	3.1
vortex	7	20	2.9
vpr	10	28	2.8
wupwise	16	36	2.3
average	14	30	2.1

Table 6.4: The average dynamic block sizes for the SPEC2000 benchmarks when compiled with basic blocks (-O3) and hyperblocks (-O4).

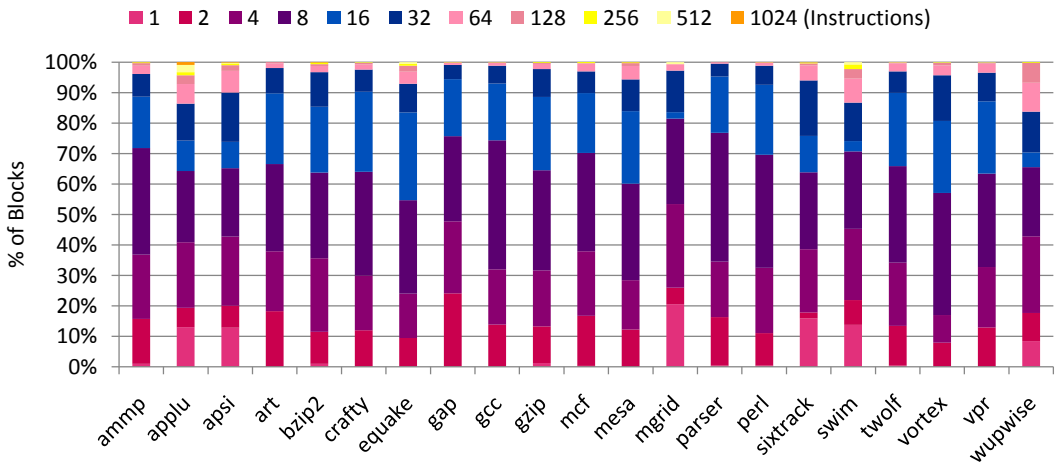


Figure 6.2: Breakdown of the block sizes when compiling to the TRIPS ISA with support for 1024 instruction blocks.

between 96-100% of all blocks formed by the compiler. While 64 instruction blocks capture 93-100% and 32 instruction blocks 84-99%. The results show that the current 128 instruction limit is aggressive for the percentage of blocks that require 128 instructions, however 64 instruction blocks would be too small. Since there is a range of block sizes produced for each benchmark, future EDGE architectures may want to support variable sized blocks to allow the compiler to take advantage of this variation.

6.2.2 Load-Store Identifiers

Load-store identifiers limit the number of memory instructions the compiler can place in a block, which affects the blocks produced by the code generator, the hyperblock generator, and the register allocator during spilling. To

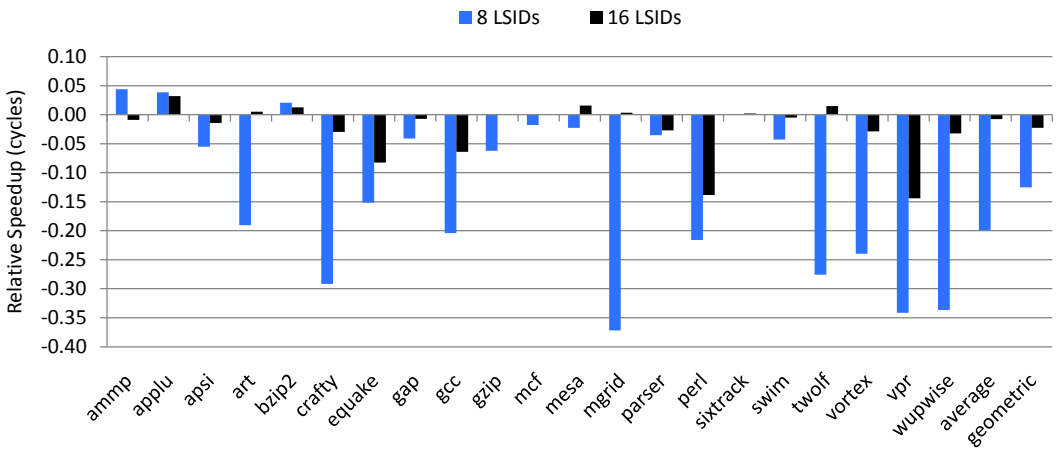


Figure 6.3: Speedup comparing 8 and 16 load-store identifiers to 32 LSIDs.

quantify the impact on performance of the load-store identifier constraint we compiled each benchmark using a maximum of 8 and 16 LSIDs then ran the resulting binaries on the TRIPS prototype.

Figure 6.3 shows the speedup in cycles compared to a baseline of 32 load-store identifiers. Using 16 LSIDs causes the benchmarks to slowdown between 1-14% with an average slowdown of 1%. When utilizing eight LSIDs the slowdown ranges from 2-37% with an average slowdown of 20%. Performance on a whole does not degrade terribly with 16 LSIDs but equake, gcc, perl, and vpr all benefit from the larger 32 LSIDs limit. Eight identifiers though is too restrictive with substantial performance loss.

Next, we performed the same experiment for load-store identifiers as we did for block size to explore the distribution of identifiers when compiling to an hypothetical ISA that supports an unlimited number of LSIDs. Figure 6.4

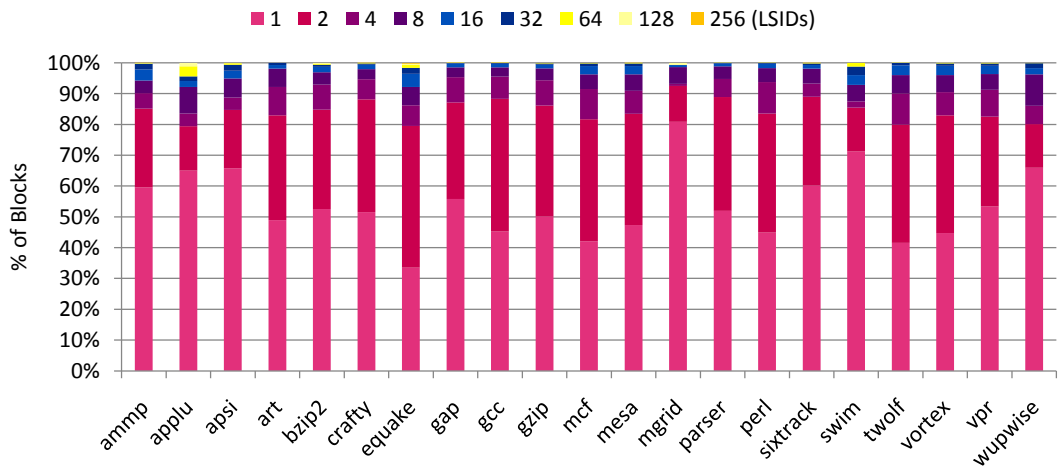


Figure 6.4: Breakdown of the load-store identifiers used when compiling to the TRIPS ISA with support for 256 load-store identifiers.

gives the results. Using 32 load-store identifiers captures between 96-100% of the blocks produced. While 16 LSIDs capture between 94-100% and 8 LSIDs between 92-99% of the blocks. Very few blocks require more than 32 LSIDs and these results support 32 LSIDs as an appropriate number.

6.2.3 Register Spills

We evaluate the number of spills when using 128 registers, and the number of iterations of the register allocator due to splitting blocks with spills. When the register allocator cannot assign a register to a live range the live range must be spilled. The choices of which live ranges to spill affects program performance because registers have lower access times than memory. Spilling also increases the size of blocks and can result in blocks being split when they

Benchmark	Shortest		Shortest+Size	
	Live ranges	Spills	Live ranges	Spills
ammp	2	40	2	6
applu	184	642	187	500
apsi	21	98	21	77
equake	6	39	6	18
mesa	99	429	101	254
mgrid	3	17	3	9
sixtrack	18	156	23	81

Table 6.5: The benchmarks with spills, the number of live ranges spilled, and the total number of load and store instructions inserted when using a policy that assigns shortest live ranges first versus a policy that prioritizes live ranges based on block size.

exceed the block size or load-store identifier constraint.

Only seven of the 21 benchmarks have spills. Table 6.5 gives the number of spills using two assignment policies. The first policy orders live ranges based on their length and assigns the shortest live ranges first (shortest). The second policy orders live ranges by their length and by the number of instructions in the blocks that use or define the live range (shortest+size). The column labeled "live ranges" gives the number of live ranges in each benchmark that were spilled. The column labeled "spills" gives the total number of load and store instructions inserted to spill the live range. These results show that the two heuristics spill almost the same number of live ranges, however the policy that accounts for block size inserts half the number of load and store instructions. The fewer instructions inserted, the less likely the block splitter must run.

Benchmark	Functions		Iterations		
	Total	w/ Spills	Min	Mean	Max
ammp	181	1	1	1.0	1
applu	16	7	1	1.6	3
apsi	97	4	1	1.3	2
equake	28	1	2	2.0	2
mesa	1107	4	1	2.0	3
mgrid	12	3	3	3.0	3
sixtrack	241	4	1	1.8	3

Table 6.6: The number of functions with spills along with the number of times the register allocator must run because of spilling using an assignment policy that accounts for block size.

We use a global register allocator (i.e. register allocation is performed once per function). If there are no spills, or any blocks with spills are legal, then register allocation is complete. However, when blocks with spills are illegal the function must be reallocated. Table 6.6 shows the total number of functions in each benchmark, and the number of functions that have spills using the assignment policy that assigns live ranges based on length and block size. The table gives the minimum (min), mean, and maximum (max) number of times the register allocator ran for the functions with spills. When the register allocator runs once and completes that is counted as one iteration in the table. The results show that the number of functions with spills is low, and that when the register allocator does spill, the allocator never runs more than three times.

Benchmark	128 Registers (cycles)	64 Registers (increase)	32 Registers (increase)
ammp	111294525	1.00	0.99
applu	29800174	1.03	1.06
apsi	99378361	1.00	1.00
crafty	267076938	-	0.99
equake	915040171	1.05	1.01
gap	164267736	-	1.00
gcc	329818233	-	1.00
gzip	830679466	-	1.01
mesa	8065919659	0.89	0.89
mgrid	9299123735	1.04	3.59
parser	619210532	-	1.01
perl	667597502	-	1.12
sixtrack	5511148173	1.17	1.03
swim	62525435	-	1.01
twolf	208664357	-	1.03
vortex	308241323	-	1.02
vpr	30112006	-	1.11
wupwise	13890658362	-	0.98

Table 6.7: The change in cycles for the benchmarks with spills when using 32 and 64 registers compared to 128 registers.

6.2.4 Register File Size

To measure the affects of the register file size on program performance, each benchmark was compiled with 32 and 64 registers. The binaries were then run on the TRIPS prototype and compared against the cycle counts for 128 registers. Table 6.7 shows the number of cycles for 128 registers and the increase or decrease in cycles when changing the register file size.

When using 64 registers, seven benchmarks have spills (the same bench-

marks that spill with 128 registers). Performance with 64 registers is similar to 128 registers. Mesa though is 11% faster, while sixtrack is 17% slower. The speedup in mesa is due to a 50% decrease in L1 instruction cache misses. While sixtrack's slowdown is because of a 2x increase in pipeline flushes due to dependence violations from the spill instructions.

With 32 registers eleven additional benchmarks have spills (18 total), however the number of additional spills is small, and in most cases performance matches 128 registers. Exceptions though are applu, mesa, mgrid, perl, and vpr, which are all slower when using 32 registers. Performance for mgrid is notably worse with 32 registers. With 128 registers mgrid already suffers from high register pressure and reducing the register file size to 32 causes mgrid to take almost four times longer to execute. There are 32% more block commits, 55x the number of block flushes, and four times as many blocks fetched due to spilling.

6.2.5 Nullification

The constant output constraint requires that all paths of execution through a block produce the same set of register writes and store identifiers. Recall that the ISA provides a null instruction to support this requirement. Write instructions are nullified by inserting a single null, and store instructions are nullified by inserting a null instruction along with a "dummy" store that is assigned the LSID to nullify. These instructions add overhead to the block by utilizing space that could otherwise be used for compute instructions. In

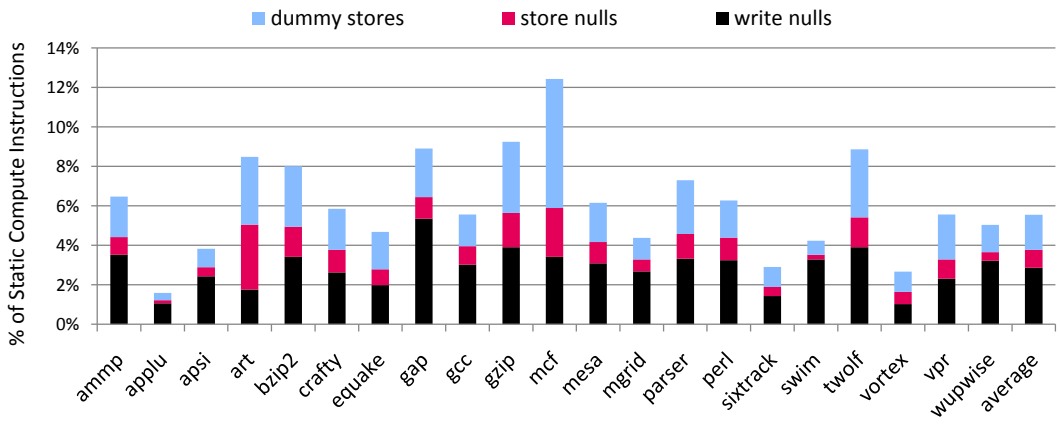


Figure 6.5: The percentage of static compute instructions required for store and write nullification.

this section we quantify the amount of overhead.

Static Overhead: Figure 6.5 shows the number of instructions added by the compiler to nullify register writes and store instructions as a percentage of the total number of *static* compute instructions. We calculate the percentages from the instructions in the TIL files for each benchmark before scheduling. These numbers exclude fanout instructions that will be inserted by the scheduler and are thus overly conservative since the additional moves will increase the number of non-null compute instructions lowering the percentages. Additionally, these numbers do not include any libraries or runtime sources.

The number of null instructions required to support write nullification ranges from between 1.07% and 5.35% of the average static instructions per

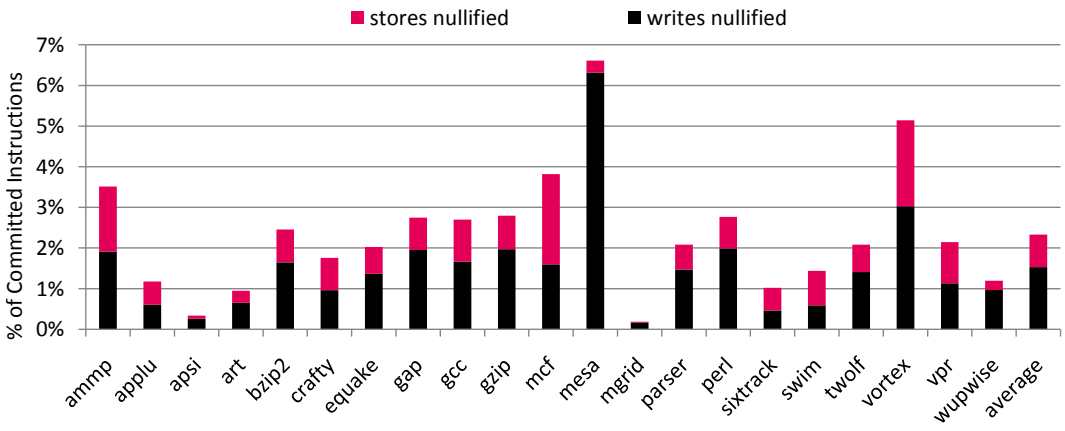


Figure 6.6: The dynamic percentage of nullified stores and nullified register writes. Every nullified store is a "dummy" store that also requires a null instruction. Every nullified register write represents a single null instruction.

block. While for store nullification the compiler inserted nulls are on average 0.15% to 3.29% of the instructions per block, and the "dummy" stores are an additional 0.37% to 6.54%. In total the overhead for nullification is between 1.59% and 12.43% when measured statically and accounts for 4 instructions or less on average.

Dynamic Overhead: Figure 6.6 gives the number of executed (non-speculative) "dummy" stores instructions and nulls for register writes. In the worst case there will also be one null instruction executed for each dummy store, although the compiler is often able to use the same null for multiple stores. The number of dummy store instructions nullified ranges from between 0.02% and 2.22% of the total dynamic instruction mix. While 0.16% to 6.32% of the dynamic instructions are nulls to support write nullification. In total the

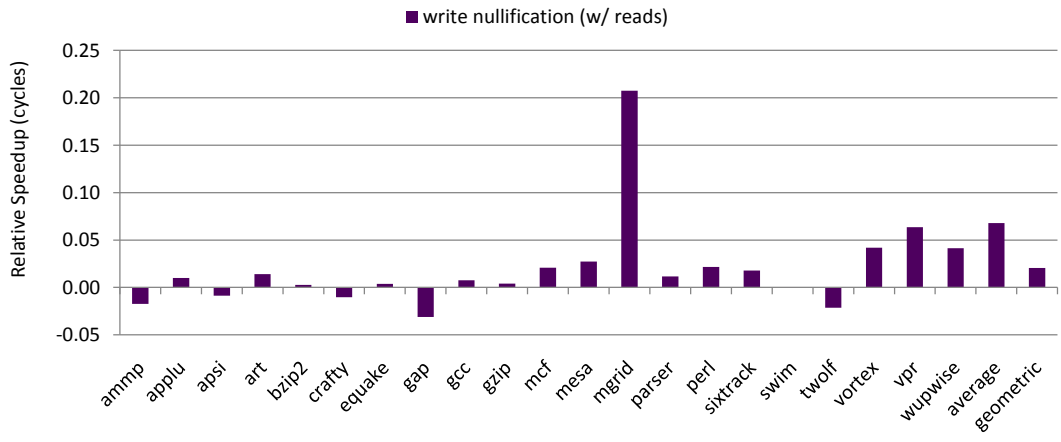


Figure 6.7: Speedup when using write nullification normalized against a baseline of reading in the register and forwarding the original value to the write.

dynamic overhead for nullification is between 0.19% and 6.61% with an average of 2.33%. In terms of actual instructions, the average number of committed dummy stores and nulls for register writes per block is 2 instructions or less.

The static overhead of nullification is on average double the dynamic instruction overhead (2 versus 4 instructions). When accounting for the additional (worst case) number of null instructions required for store nullification, the average dynamic overhead is closer to 3 instructions per block.

Forwarding the Original Register Value: Instead of nullifying write instructions, the compiler can read in the original register value and write this value out on the path that would otherwise require a null. Reading in the original value trades instruction overhead (by reducing in the number of null and fanout instructions inserted), for register pressure (due to the

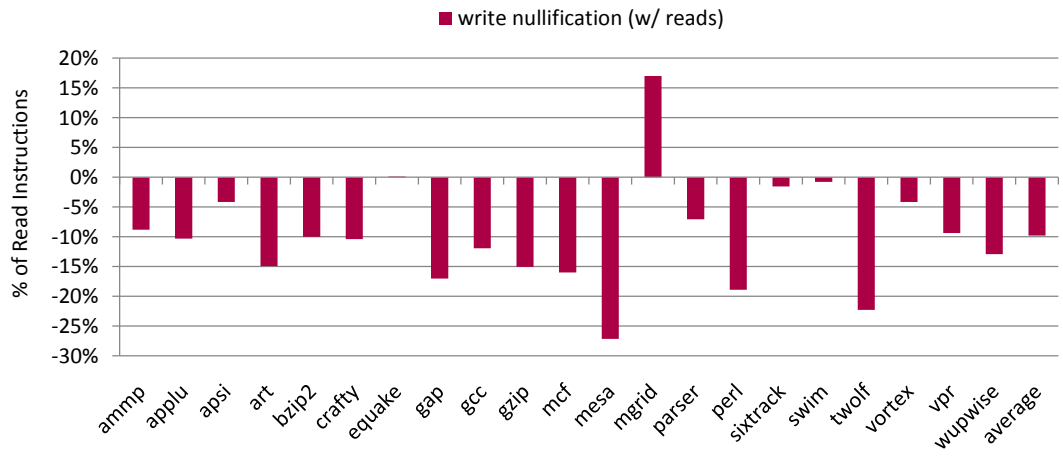


Figure 6.8: Reduction in register reads when using write nullification.

increased number of register reads required). Figure 6.7 gives the speedup (cycles) when using write nullification compared to a baseline of reading in the original value. Most benchmarks have low enough register pressure (Table 6.7) that reading in the original value does not severely degrade performance. However, write nullification does provide on average a 7% reduction in cycles (2% geometric). Mgrid especially benefits from write nullification because without it the compiler is forced to spill from the increased register pressure. Spilling results in a 30% increase in the number of double-word load instructions and a 119% increase in the number of double-word store instructions for a total increase in executed instructions of 37%.

Figure 6.8 gives the reduction in read instructions when using write nullification compared to reading in the original register value. Write nullification reduces up to 27% of the register reads from the benchmarks with an

average reduction of 10%. Mgrid has an increase in the number of read instructions when using write nullification due to the compilers ability to increase the number of instructions per block, which increases the number of register reads. Write nullification thus reduces reads to the power-hungry shared register file.

6.3 Performance

This section evaluates the performance of optimizing predicated blocks against a baseline of basic blocks (-O3) using the SPEC reference inputs with the following four optimizations:

- *hyperblocks*: Enables hyperblock formation but no additional predicate optimizations.
- *top*: Enables hyperblock formation and predicate fanout reduction using the tops of dependence chains.
- *bottom*: Enables hyperblock formation and predicate fanout reduction using the bottoms of dependence chains. Loads though are not allowed to execute speculatively.
- *bottom+speculative loads*: Enables hyperblock formation, predicate fanout reduction (bottom), and allows loads to execute speculatively.

We discuss the geometric speedup in this section for the results shown in Figure 6.9. Both hyperblock formation and predicate fanout reduction (top)

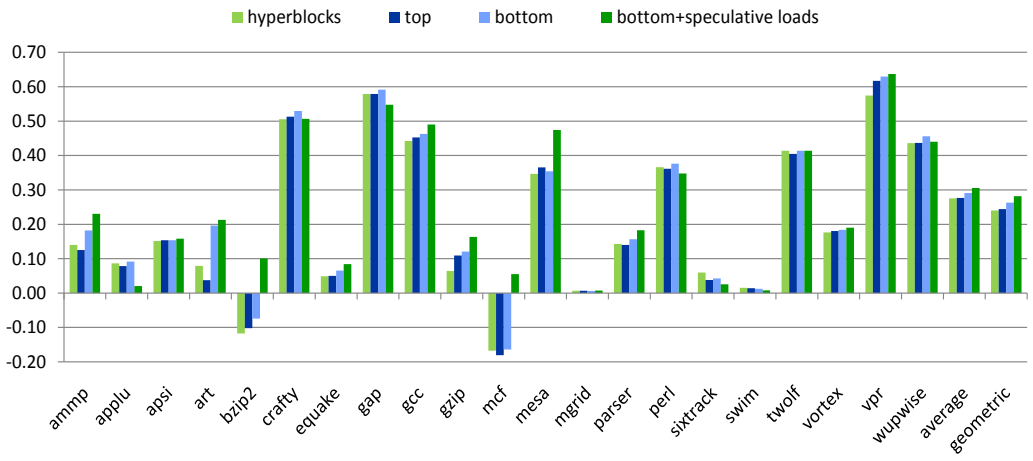


Figure 6.9: Speedup.

provide on average a 24% increase in performance. Predicating the bottoms of dependence chains increases performance 2% on average over hyperblocks alone to 26%. Bottom compared to top provides 1-2% improvement in performance except on ammp and art where bottom outperforms by 5% and 16% respectively. An advantage though of bottom is that loads can execute speculatively, which increases performance 4% on average over hyperblock formation alone to 28%. Performance on ammp, bzip2, gcc, mcf, and mesa all have marked increases in performance when loads execute speculatively. The benefits of predication are multifold: increased instruction window utilization, reduction in the number of blocks (static) executed, reduced branch mispredictions, and improved I-cache performance.

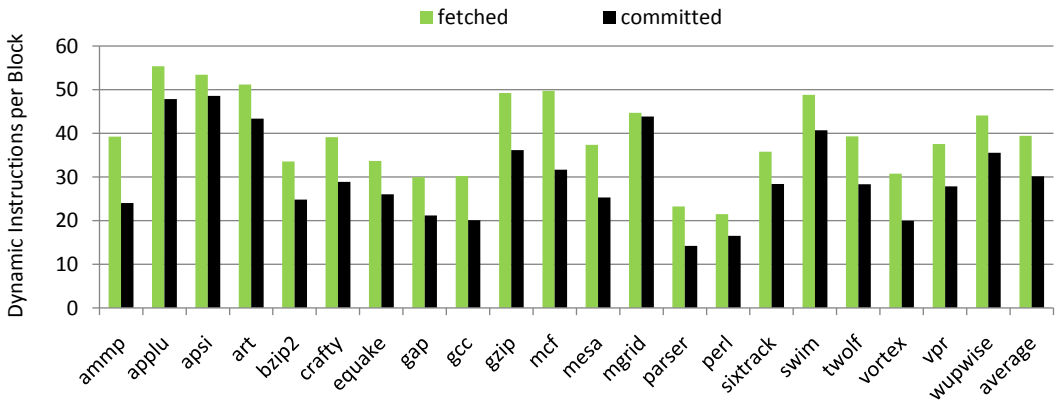


Figure 6.10: Fetched and committed instructions per block (bottom with speculative loads).

Top and bottom tradeoff energy utilization for speculative execution. One way to measure this tradeoff is to examine the difference between fetched and committed instructions. The number of instructions executed using top will be similar to the number of instructions committed. The instructions executed by bottom will be closer to the number of fetched instructions due to speculation.

Figure 6.10 gives the number of fetched and committed instructions per block for bottom (with speculative loads), and Figure 6.11 shows the same results for top. These results are for the MinneSPEC inputs. Both fetch on average 40 instructions per block. Bottom commits 30 instructions per block and top commits 28 instructions per block. This difference in committed instructions signifies that bottom is able to reduce the predicate fanout more than top, enabling additional hyperblock formation. The difference between

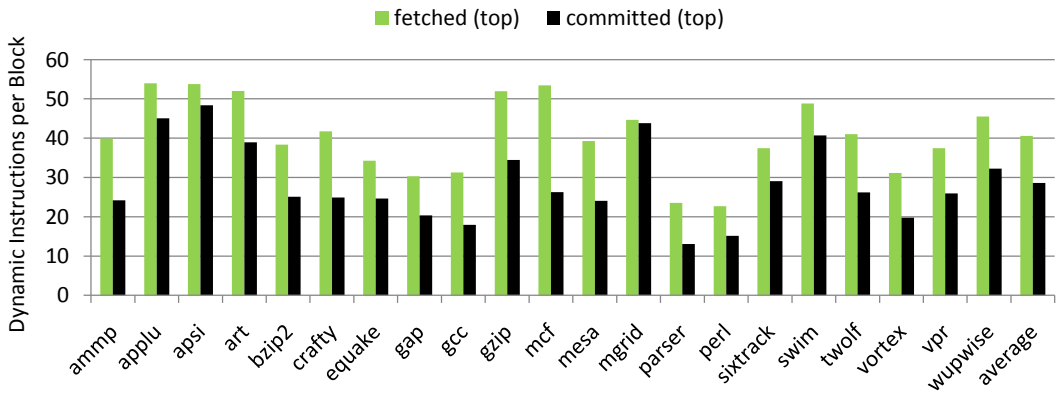


Figure 6.11: Fetched and committed instructions per block (top).

fetched and committed instructions ranges from 1-18 instructions for bottom and 1-17 instructions for top.

6.4 Summary

This chapter presented results to evaluate the complexity of the algorithms developed in this thesis, the tension between the compiler and the block constraints of the TRIPS ISA, and the performance of the current compiler flow when compiling to the TRIPS prototype processor. The results show that the compiler is meeting the new challenge of compiling to the TRIPS EDGE ISA, and achieves 28% geometric speedup on average with full optimizations compared to a baseline of basic blocks alone.

Results support that the 128 instruction limit is an appropriate block size, but future work must focus on increasing the number of instructions per

block to effectively utilize the entire instruction window. Improvements may come from additional work on the hyperblock generator, such as merging predicate blocks or individual instructions, or from classical VLIW optimizations to increase the amount of instruction level parallelism. Fewer than 32 load-store identifiers is insufficient to support the number of memory instructions per block, while results show that more than 32 LSIDs would over-provision the resource. Additionally, 128 global registers is too many and could be reduced to 64. Next generation EDGE ISAs may want to explore ways to remove or reduce the block constraints. For example, supporting variable size blocks, or relaxing the constraints on load-store identifier sharing through the use of unordered load-store queues [66]. Removing the register banks and allowing every block to read and write every global register would also reduce the constraints on the register allocator and scheduler.

Chapter 7

Conclusion

Explicit Data Graph Execution (EDGE) architectures renegotiate the boundary between hardware and software to expose and exploit concurrency. EDGE architectures utilize a block-atomic execution model in which instructions communicate directly and execute in dataflow order. 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.

Dataflow predication exploits ISA features, microarchitectural mecha-

nisms, and compiler algorithms to reduce predication overheads in an EDGE ISA while maintaining low-complexity out-of-order issue. In VLIW architectures, the execution overhead of falsely predicated instructions limits the compilers ability to perform aggressive predication. In superscalar architectures, the hardware complexity and ISA encoding difficulties inhibit the incorporation of full predication. Dataflow predication avoids both of these limitations, while reducing the predicate encoding space consumed to two bits per instruction. However, dataflow predication incurs the costs of fanning out predicates to many consumers.

7.1 Moving Forward

The TRIPS compiler project was a first step in developing the compiler support for EDGE ISAs. Much of our research has focused on the algorithms for forming legal blocks with respect to the block constraints to achieve correct execution. When we first began the TRIPS project it was not clear where the block constraints should be fixed. We knew we wanted to be aggressive but at the same time realistic. The results from this research show that we came close. However, the 128 instruction limit has been challenging for the compiler to meet and additional research is needed to move the compiler closer to producing full blocks of useful instructions. Future research directions thus center on improving the existing compiler phases, developing new optimizations that utilize the dataflow predication model, and refining the ISA to reduce the overhead of the block model.

7.1.1 Compiler Opportunities

Future compiler opportunities in general are all geared towards increasing block size and improving the efficiency of the executed code. The first area of improvement is the hyperblock generator. The back end hyperblock generator was developed after most of the other back end compiler phases. Out of convenience we used the same intermediate representation that was developed for supporting the block constraints–namely the block flow graph with individual predicate flow graphs for each block. What we found though is that blocks are too coarse a granularity for effective hyperblock formation. If the hyperblock generator instead selected individual instructions to if-convert and merge (or even predicate blocks), this would allow for a tighter packing, which should translate into an increase in the number of instructions per block and a reduction in the number of executed blocks.

There are other optimizations that can work synergistically with hyperblock formation to enable additional block merging. Any structural hazard in the program limits the hyperblock generators ability to merge blocks (i.e., function calls, loop back edges, and merge points). Function calls can be eliminated by aggressive inlining. We do support function inlining but good heuristics are notoriously hard to develop. If inlining were performed in the back end, knowledge of block constraints could be used to drive the inliner. There are a variety of simple loop transformations that can be applied in the back end to reduce or eliminate back edges. Loops with known bounds can be flattened and completely removed from the program. Iterations from loops

can be peeled, or loops whose bodies are contained within a single block could be unrolled until one of the block constraints was reached. More sophisticated loop transformations derived from work on automatic vectorization [3] could be applied to reduce the number of loops. Finally, merges can be eliminated with tail duplication. Once the compiler produces denser blocks, a next step is to improve the fraction of useful instructions in blocks. Edge or path profiling could guide the hyperblock generator when selecting regions for inclusion.

7.1.2 Architectural Refinements

Any block constraint that can be reduced or eliminated will improve the compilers ability to form blocks. Having a large fixed block size is currently the the most challenging constraint for the compiler. If the ISA supported variable size blocks, the compiler would have additional freedom when forming blocks, resulting in a higher likelihood of utilizing the entire instruction window. The register constraints also limit the compiler. Even though TRIPS supports 128 global registers, each block can only read and write 32 registers (eight reads or writes from four banks each). These restrictions were necessary to support a large register file, but results from this research show that a smaller register file of 64 registers performs equally well. If the register file size is reduced then both of these constraints could be eliminated.

The limited encoding space in instruction formats for instruction targets requires the compiler to build fanout trees to distribute operands to their consumers. Fanout instructions introduce overhead, as they occupy space in

the block that could be used for compute instructions, and increase the dependence height on the critical path. Using a broadcast network to distribute operands that have a high degree of fanout may be one solution. Instruction formats already contain a target field which encodes the target of the operand (either left, right, or predicate operand or global register write). The target field could be expanded to encode a broadcast channel identifier. Then a field to select the broadcast channel to receive an operand on could be added to each instruction format. Preliminary research has shown that broadcast channel support in EDGE ISAs utilize lower energy than the equivalent fanout tree, and provide a slight improvement to performance [39].

The biggest overhead of predication may eventually be the fraction of mispredicated instructions in the window, which reduce the effective window size. At any given moment in a program s execution, there are three classes of instructions in the window: useful instructions that are correctly predicated, useless instructions that are falsely predicated, and instructions past a branch misprediction, all of which are useless. Each window size has a sweet spot between no predication (pure superscalar) and all predication (pure dataflow) for maximum parallelism. If instruction window sizes continue to increase, however, the relative costs of increased predication will continue to decline, pushing the ideal balance toward more aggressive predication. It is possible that the long term solution to branch mispredictions will not be more accurate predictors, but conversion of most unpredictable branches to predicates in extremely large instruction windows. For this solution to be viable, some form

of predicate predication [18] will likely be necessary to reduce the increases in dependence heights, caused by predication, down the true paths of execution.

With the end of technology scaling in sight, future architectures must strive to improve energy efficiency by exploiting parallelism. The compiler will play an even more crucial role as these architectures re-negotiate the boundary between hardware and software. ISA's such as EDGE, which build upon the strengths of both hardware and software, already offer one potential solution.

Appendices

Appendix A

TRIPS Application Binary Interface

This appendix describes the application binary interface for the TRIPS prototype processor. The goal of this document is to provide a consistent standard for vendors and researchers to follow. No thought has been given to any other language besides C and FORTRAN. You are encouraged to build upon and expand this document for other languages such as C++ and Java. For additional information relevant to the TRIPS Application Binary Interface, please consult the following manuals:

- *TRIPS Processor Reference Manual*
- *TRIPS Intermediate Language (TIL) Manual*
- *TRIPS Assembly Language (TASL) Manual*
- *TRIPS Object File Format (TOFF) Specification*

A.1 Architectural Description

For a complete architectural description, refer to the *TRIPS Processor Reference Manual*.

A.1.1 Registers

The TRIPS architecture provides 128 general purpose registers (GPRs). By convention GPRs are named R0 - R127. The architecture makes no distinction between floating point and general purpose registers. The TRIPS architecture does not define any special purpose control registers which are accessible through the instruction set.

A.1.2 Fundamental Types

Table A.1 shows the TRIPS equivalents for ANSI C fundamental types along with their sizes and alignments. Fundamental types are always aligned on natural boundaries. The TRIPS architecture supports 64, 32, 16 and 8-bit load and store operations. All data is in big endian byte order. For the purposes of this document, we define the following types:

- *doubleword* – A doubleword is 64-bits and the least significant 3-bits of the address of a doubleword in memory are always zero.
- *word* – A word is 32-bits and the least significant 2-bits of the address of a word in memory are always zero.
- *halfword* – A halfword is 16-bits and the least significant bit of the address of a halfword in memory is always zero.
- *byte* – A byte is 8-bits.

ANSI C	**Size (bytes)**	**Alignment (bytes)**
char	1	1
unsigned char	1	1
signed char	1	1
short	2	2
unsigned short	2	2
signed short	2	2
int	4	4
unsigned int	4	4
signed int	4	4
enum	4	4
long	8	8
unsigned long	8	8
signed long	8	8
long long	8	8
unsigned long long	8	8
signed long long	8	8
float	4	4
double	8	8
long double	8	8

Table A.1: TRIPS Fundamental Types

Compound Type	Alignment
Arrays	Same as individual elements
Unions	Most restrictive alignment of members
Structures	Same as unions
Bit fields	Same as individual elements

Table A.2: Alignment of Compound Types

A.1.3 Compound Types

The alignment requirements for arrays, structures, unions and bit fields are summarized in Table A.2. Arrays are aligned according to the alignment of their individual elements. For example,

```
char ac[10]; /* aligned on 1-byte */
short as[10]; /* aligned on 2-bytes */
float af[10]; /* aligned on 4-bytes */
```

Structures and unions are aligned according to their most restrictive element. Padding should be added to the end of the structure or union to make its size a multiple of the alignment. Fields within structures and unions are aligned according to the field's type with the exception of bit fields. Padding should be added between fields to ensure alignment. For example,

```
struct s1 {
  char bc[9];  /* aligned on 1-byte */
  short bs;    /* aligned on 2-bytes */
  int bi;      /* aligned on 4-bytes */
```

```
  char bc2[9]; /* aligned on 1-byte */
};
```

The individual elements `bc` and `bc2` are aligned on 1-byte boundaries. The elements `bs` and `bi` are aligned on a 2-byte and 4-byte boundaries respectively. A 1-byte pad will be added between `bc` and `bs` in order to align `bs` on a 2-byte boundary. Since `int` is the most restrictive element of the structure, a 3-byte pad would be added to the end of the structure to align it on a 4-byte boundary.

The maximum size of a bit field is 64-bits. Bit fields cannot be split over a 64-bit boundary. Zero-width bit fields pad to the next 32-bits, regardless of the type of the bit field. No other restriction applies to bit field alignment. However, bit fields impose alignment restrictions on their enclosing structure or union according to the fundamental type of the bit field.

A.2 Function Calling Conventions

A.2.1 Register Conventions

Table A.3 defines the register conventions for the TRIPS architecture. There is no distinction between floating point and integer values for the purpose of the conventions.

Registers R0, R1 (stack pointer), R2 (return address) and R12–R69 are *callee-save* or *non-volatile*, which means that the compiler preserves their values across function calls. Any function which uses any register in this class

Register	Usage and Description	Lifetime
R0	System Call Identifier for SCALL	Callee-save
R1	Stack Pointer	Callee-save
R2	Return Address Register	Callee-save
R3	Arguments and Return Values	Caller-save
R4	Arguments and Return Values	Caller-save
R5 - R10	Arguments	Caller-save
R11	Reserved for Environment Pointer	Caller-save
R12	Frame Pointer or Local Variable	Callee-save
R13 - R69	Local Variables	Callee-save
R70 - R127	Local Variables	Caller-save

Table A.3: Register Conventions

must save the value before changing it, and restore it before the function returns.

The remaining registers, R3–R11 and R70–R127, are *caller-save* or *volatile*, which means that they can be overwritten by a called function. The compiler will ensure that any function which uses any register in this class must save the value before calling another function, and restore it after that function returns, if that value is to be reused after the call.

Register R1 (SP) contains the function's stack pointer. It is the responsibility of the function to decrement the stack pointer by the size of its stack frame upon entry in the function *prologue* and increment the stack pointer by the size of its stack frame upon exit in the function *epilogue*. To support the debugger, the compiler stores the caller's stack pointer in the link area as a *back chain pointer*, prior to decrementing the stack pointer register (SP) in the prologue.

If a function uses *alloca*, which allocates space for the user on the stack, register R12 (FP) is used to access the function's stack frame while allowing the stack pointer (R1) to be changed by *alloca*. Upon entry to such a function, the address in R1 is first decremented and then this address is copied into R12. Then register R12 is copied back into R1 just before register R1 is incremented on the function's return.

Register R2 contains the function's return address upon entry. It is the responsibility of the function to preserve its return address so that it may return to its caller. If the function calls no other functions, it may do this by keeping its return address in R2. Otherwise, it must save the return address in the link area.

A.2.2 Stack Frame Layout

Each function has a stack frame on the runtime stack which grows downward from high addresses. Figure A.1 shows the stack frame organization. Note that the figure shows low memory addresses at the top and high addresses at the bottom. From low to high addresses, the stack frame for a function (callee) contains:

- Fixed Size Link Area
- Argument Save Area
- Local Variables

- Register Save Area

A.2.2.1 Link Area

This fixed size area holds (a) the address of the caller's stack frame and (b) the callee's return address (Figure A.2):

- The first doubleword (lowest address in the callee's stack frame) contains the caller's stack pointer value, sometimes called the "back chain". The first stack frame (that is, the stack frame of the *_start* function) will have a back chain value of 0.

- The second doubleword contains the callee's return address, which is set by the caller before branching to the function. If debugging is not required, this doubleword may be left undefined in order to avoid a store to memory.

  If a function dynamically allocates space on the stack (e.g., *alloca*()), then the allocated space must be between the link area and the argument save area. This means that the link area must be moved when the allocation is performed. The stack pointer register must always point to the link area.

Figure A.3 shows the use of back chain pointers to traverse the stack frames.

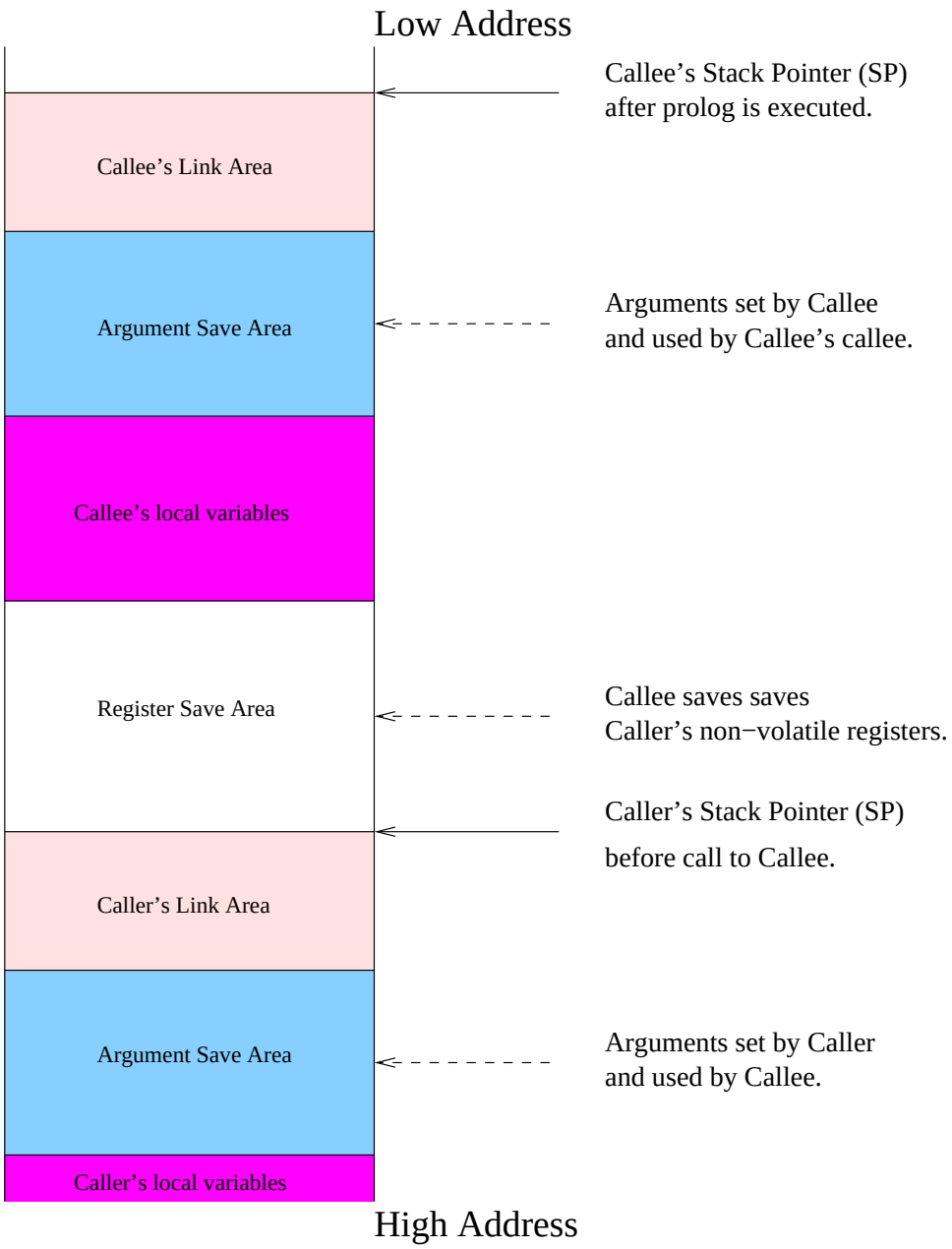


Figure A.1: Stack Frame Layout

(SP)	Back chain pointer (i.e., stack frame address of caller)
$(SP)+8$	Callee's return address

Figure A.2: Link Area After Callee Prologue

A.2.2.2 Argument Save Area

This variable size area is large enough to hold all of the arguments that a routine may pass to any of the routines that it calls as determined by:

- A minimum of `MAX_ARG_REGS` (8) doublewords is usually reserved for the argument save area because the caller can not know if it is calling a routine that uses `va_start`. See section A.2.5.
- For a "leaf routine" this area may contain 0 doublewords. When a routine calls a function it places the first `MAX_ARG_REGS` doublewords of arguments in the argument registers (R3 ... R10). Any additional doublewords of arguments are placed starting in doubleword 8 of the argument save area. Each argument is placed in at least one register or in at least one doubleword in the argument save area. Arguments larger than a doubleword may be split between a register and the argument save area. The least significant 3-bits of the address of any argument in the argument save area are zero.

A.2.2.3 Local Variables

Any local variables of a callee that must reside in memory are placed in the local variable area. The least significant 3-bits of the address of any

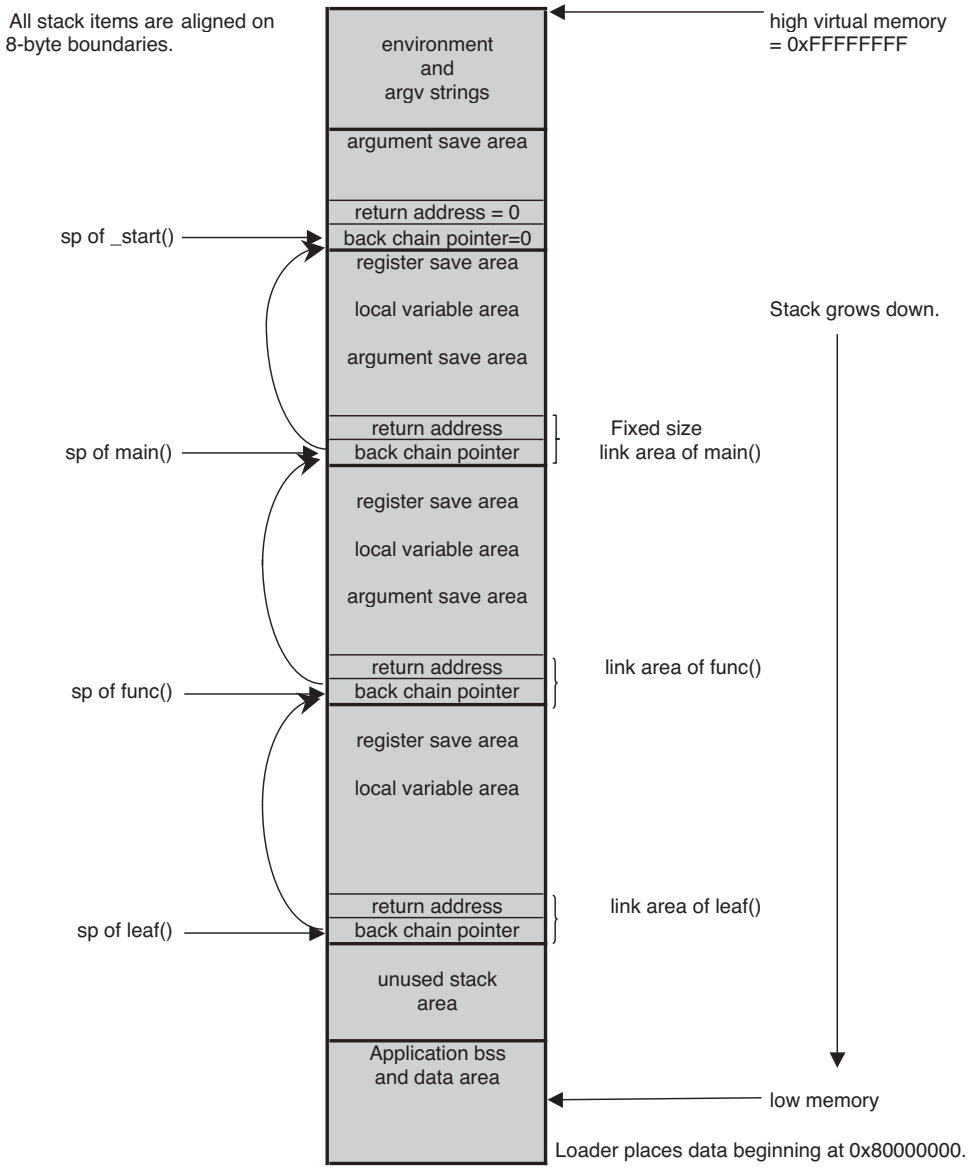


Figure A.3: TRIPS Stack Linkages

variable are always zero. The size of the area may be zero.

A.2.2.4 Register Save Area

The register save area holds the contents of any of the callee-save registers that the callee modifies. Registers are saved to increasing addresses. For example, if the callee modifies only the callee-save registers R60 and R62 then the register save area will be 16 bytes. Register R60 will be stored at offset 0 and register R62 will be stored at offset 8 into the register save area. The least significant 3-bits of the address of any register in the register save area are zero. The size of the area may be zero.

A.2.2.5 Requirements

The following requirements apply to the stack frame:

- The least significant 4-bits of the value in the stack pointer register (SP) shall always be zero.
- The stack pointer shall point to the last word of the current stack frame. Thus, (SP) is the address of the "back chain" word of the link area. The stack shall grow downward, that is, toward lower addresses.
- The stack pointer shall be decremented by the called function in its prologue and restored prior to return.
- Before a function changes the value in any callee-save general register, Rn, it shall save the value in Rn in the register save area.

A.2.3 Parameter Passing

Both scalar and compound type parameters are passed in registers R3 through R10. Parameters shall be assigned consecutively to registers so that R3 contains the first function parameter. Assuming that the first argument is 8 bytes or less, R4 contains the second. This continues until all argument registers are occupied. If there are not enough registers for the entire parameter list then the parameters overflow in consecutive order onto the argument save area of the stack.

Scalars less than 64-bits are right justified within the register. The caller must not assign more than a single scalar argument to a register.

Compound types (C structs) larger than 64-bits are packed into consecutive registers. Compound types less than 64-bits are placed within the register in the position that allows a simple store to place them in memory aligned upon a doubleword boundary (see Figure A.4).

The argument save area is located at a fixed offset of `ARG_SAVE_OFFSET` (24) bytes from the stack pointer, and is reserved in each stack frame for use as an argument list. A minimum of `MAX_ARG_REGS` (8) doublewords is reserved if the routine calls another routine. The size of this area must be sufficient to hold the longest argument list being passed by the function which owns the stack frame. Although not all arguments for a particular call are located in storage, consider them to be forming a list in this area, with each argument occupying one or more doublewords. If more arguments are passed than can

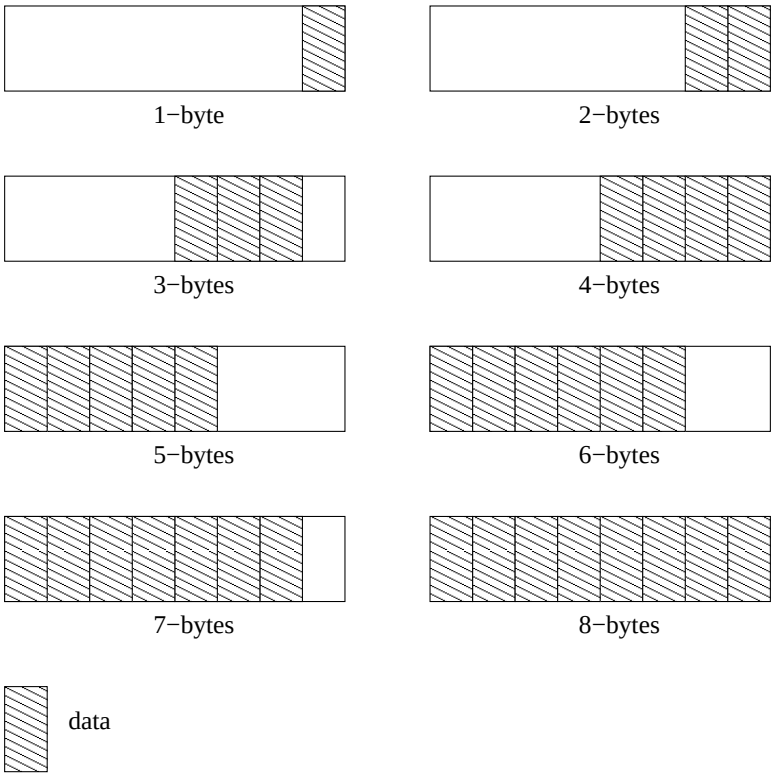


Figure A.4: Passing C Structs

be stored in registers, the remaining arguments are stored in the argument save area.

The rules for parameter passing are as follows:

- Each argument is mapped to as many doublewords of the argument save area as are required to hold its value.
    1. Single precision floating point values are mapped to a single doubleword.
    2. Double precision floating point values are mapped to a single doubleword.
    3. Simple integer types (char, short, int, long, enum) are mapped to a single doubleword. Value shorter than a doubleword are sign or zero extended as necessary.
    4. Pointers are mapped to a single doubleword.
    5. Aggregates and unions passed by value are mapped to as many doublewords of the argument save area as the value uses in memory.
    6. Other scalar values, such as FORTRAN complex numbers, are mapped to the number of doublewords required by their size.
- If the callee has a known prototype, arguments are converted to the type of the corresponding parameter before being mapped into the parameter save area. For example, if a long is used as an argument to a float double

parameter, the value is converted to double-precision and mapped to a doubleword in the argument save area.

- The first `MAX_ARG_REGS` (8) doublewords mapped to the argument save area are never stored in the argument save area by the calling function. Instead, these doublewords are passed in registers as described above.

- Argument values beyond the first eight doublewords must be stored in the argument save area following the first eight doublewords. The first eight doublewords in the argument save area are reserved for the initial arguments, even though they are passed in registers.

- General registers are used to pass some values. The first eight doublewords mapped to the argument save area correspond to the register R3 through R10. If the arguments are mapped to fewer than eight doublewords of the argument save area, registers corresponding to those unused doublewords are not used.

- If the callee takes the address of any of its parameters that are passed in registers, then those parameters must be stored by the callee into the argument save area.

Note: if the compilation unit for the caller contains a function prototype, but the callee has a mismatching definition, and if the callee takes the address of any of its parameters, the wrong values may be stored in the first eight doublewords of the argument save area.

A.2.4 Return Values

Functions shall return values of type float, double, int, long, enum, short, and char, or a pointer to any type, as unsigned or signed integers as appropriate, zero- or sign-extended to 64-bits if necessary, in R3.

Aggregates or unions of any length shall be returned in a storage buffer allocated by the caller. The caller will pass the address of this buffer as a hidden first argument in R3, causing the first explicit argument to be passed in R4. This hidden argument is treated as a normal formal parameter, and corresponds to the first doubleword of the parameter save area.

Functions shall return complex floating point scalar values of size 16-bytes or less in registers R3 (real-part) and R4 (imaginary part).

A.2.5 Variable Arguments

If the callee uses `va_start` it is the callee's responsibility to store the registers R3 through R10 in the argument save area. The remaining arguments are stored by the caller.

The `va_start` operation causes the address of the specified parameter to be stored in the doubleword allocated for the `va_list` variable. As each argument is accessed by `va_arg` this address is incremented by the proper multiple of 8. There is no provision in this specification that defines how a "variable argument" function can determine the number of arguments that were passed to it.

A.3 Runtime Support Functions

A.3.1 Application Memory Organization

The TRIPS prototype runtime system lays out virtual memory for applications from high virtual addresses to low virtual addresses as follows:

- *environment* – At the "top", or highest address, of application memory is the program environment, which is passed through to the program loader in the `**envp` string array, by the call to the program's `main()` routine.
- *stack* – Beneath the program environment area is the stack, which grows "downward" in 8-byte decrements, toward lower addresses.
- *heap* – The heap, placed on top of the program's text and data segments, grows upward by means of the `brk()` system call.
- *bss* – The unitialized data section, for variables tagged with the .comm directive, sets the boundary between the program text and data area and the heap area.
- *initialized data* – The program's read/write initialized data section appears at lower addresses than the .bss area.
- *read-only data* – This area is reserved for initialized data that is marked by the compiler with the .rdata directive as read-only.
- *program text* – At the lowest program addresses are the code blocks comprising the program's executable section.

Register	Description
R1	The initial stack pointer, aligned to a 16-byte boundary.
R3	*argc*–the number of program arguments.
R4	*argv*–the array of NULL-terminated argument strings.
R5	*envp*–the array of NULL-terminated environment strings.

Table A.4: Registers Initialized by the Loader

A.3.2 Process Initialization

Application behavior at startup on a TRIPS processor is modeled on PowerPC conventions [84]. For an application whose entry point is defined as:

```
int main(int argc, char ** argv, char ** envp)
```

Table A.4 lists the contents of registers when the loader returns control to the system software. The contents of other registers are unspecified. It is the responsibility of the application to save those values that will be needed later.

The loader will push the argument count, argument values, and environment strings as the first items on the user-stack, starting at the top of application memory. Next, the loader will push the *addresses* of those strings onto the stack. Hence, R1 will point to the stack address just below the values supplied from the environment and arguments to the program, whose value is a NULL pointer.

A.3.3 System Calls

System call support on the TRIPS prototype simulators is provided through the SCALL instruction. As defined in the *TRIPS Processor Reference Manual*, when a SCALL instruction is executed, a System Call Exception will occur after the program block with the SCALL commits. The TRIPS prototype simulators provide a runtime exception handler that determines the type of system call and services the request.

To invoke a system call, the identifier for the call is placed in R0. The return address and arguments for the call are passed in R2 and R3–R10 in accordance with the *function calling conventions*, and upon completion, the result code is returned in R3. If the system call was serviced successfully, the value returned in R3 will be 0. Otherwise, R4 will contain the value of errno from the simulator's host environment. Note that if no error has occurred, the value of R4 will be undefined upon return from a system call.

The TRIPS prototype simulators currently provide support-by-proxy for the system services listed in Table A.5. These services are defined in the /usr/include/sys/syscalls.h TRIPS system header file.

A.4 Standards Compliance

This section documents any deviation from the relevant standards in use for the TRIPS system. This section discusses only known deviations for which no compliance is planned. All other deviations should be regarded as

Service	Identifier
exit	1
read	3
write	4
open	5
close	6
creat	8
unlink	10
time	13
lseek	19
brk	45
gettimeofday	78
stat	106
lstat	107
fstat	108

Table A.5: System Call Identifiers

bugs in the relevant software or hardware. The relevant standards are:

- ANSI™ X3.159-1989 1989 C Programming Language
- ISO/IEC 9899 1999 C Programming Language
- ANSI™ X3.9-1978 Fortran 77 Programming Language
- IEEE 754-1985 and IEEE 854-1987 Floating Point Representation

A.4.1 C Standards

A.4.1.1 Calling Conventions

As TRIPS does not support operations on 32-bit IEEE single precision floating point values, single precision floating point values are always passed

as double precision arguments to called subroutines. See Section 3.3.2.2 of the ANSI^{TM} X3.159-1989 standard and Sections 6.5.2.2 and 6.9.1 of the ISO/IEC 9899 standard.

A.4.2 F77 Standards

The TRIPS compiler does not support the "assigned goto" capability as specified in Section 11.3 of the ANSI^{TM} X3.9-1978 standard.

A.4.3 Floating Point Representation

See the "TRIPS Processor Architecture Manual: Version 1.2: Tech Report TR-05-19 (03/10/05)" for information on this subject.

Bibliography

[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. D. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[3] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, October 2001.

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

[5] D. I. August. Hyperblock performance optimizations for ilp processors. Master's thesis, University of Illinois at Urbana-Champaign, 1996.

[6] D. I. August. *Systematic compilation for predicated execution*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[7] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. mei W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, July 1998.

[8] D. I. August, J. W. Hwu, J.-M. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu. The program decision logic approach to predicated execution. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 208–219, May 1999.

[9] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 92–103, December 1997.

[10] M. Beck, R. Johnson, and K. Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, 1991.

[11] D. B. Behnam Robatmili, Katherine Coons and K. McKinley. Register bank assignment for spatially partitioned processors. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2008.

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

[13] G. W. Brendon, B. D. Cahoon, J. E. B. Moss, K. S. McKinley, E. J. Wright, and J. Burrill. Common language encoding form (clef) design document. Technical report, University of Massachusetts-Amherst, Technical Report 97-58, 1997.

[14] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, Jul 1998.

[15] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, October 2004.

[16] 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 the TRIPS Team. Scaling to the End of Silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

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

[18] W. Chuang and B. Calder. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 183–192, June 2003.

[19] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987.

[20] K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. S. McKinley. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2006.

[21] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, San Francisco, CA, USA, 2003.

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

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[24] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 26–38, April 1989.

[25] J. Dennis and D. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132, January 1975.

[26] J. R. Ellis. *Bulldog: a compiler for VLIW architectures (parallel computing, reduced-instruction-set, trace scheduling, scientific)*. PhD thesis, Yale University, New Haven, CT, USA, 1985.

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

[28] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the trips computer system. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1–12, March 2009.

[29] 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.

[30] T. Instruments. Texas instruments: TMS320C64x technical overview, February 2000.

[31] Intel. Intel itanium architecture software developer's manual. volume 1: Application architecture, January 2006.

[32] V. Kathail, M. Schlansker, and B. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.

[33] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Rice University, 1993.

[34] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in ssa form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.

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

[36] 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, October 2002.

[37] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, 2005.

[38] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, 2002.

[39] D. Li, B. Robatmili, M. S. S. Govindan, A. Smith, S. Keckler, and D. Burger. Compiler-assisted hybrid operand communication. Technical Report TR-09-33, The University of Texas at Austin, November 2009.

[40] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[41] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292–302, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[42] B. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

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

[44] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 238–247, October 1992.

[45] 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, December 1992.

[46] R. McDonald, D. Burger, S. W. Keckler, K. Sankaralingam, and R. Nagarajan. TRIPS processor reference manual. Technical Report TR-05-19, The University of Texas at Austin, March 2005. http://www.cs.utexas.edu/users/cart/trips.

[47] S. W. Melvin, M. C. Shebanow, and Y. N. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *MICRO 21: Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, pages 60–63, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[48] S. Microsystems. Ultrasparc iii cu user's manual, January 2004.

[49] R. Morgan. *Building an optimizing compiler.* Digital Press, Newton, MA, USA, 1998.

[50] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 235–245, 1997.

[51] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[52] 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, October 2004.

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

[54] 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.

[55] J. Park, J.-H. Lee, and S.-M. Moon. Register allocation for banked register file. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop*

on Languages, compilers and tools for embedded systems, pages 39–47, New York, NY, USA, 2001. ACM.

[56] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129, April 1994.

[57] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, 1999.

[58] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towie. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs. *Computer*, 22(1):12–26, 28–30, 32–35, January 1989.

[59] B. Robatmili, K. Coons, D. Burger, and K. McKinley. Register bank assignment for spatially partitioned processors. In *21th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC08)*, 2008.

[60] A. Rogers and K. Li. Software support for speculative loads. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, 1992.

[61] R. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[62] K. Sankaralingam. *Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities*. PhD thesis, The University of Texas at Austin, 2006.

[63] 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.

[64] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[65] S. Sethumadhavan, R. Desikan, R. McDonald, D. Burger, and S. Keckler. Design and implementation of the trips primary memory system. In *24th International Conference on Computer Design (ICCD)*, October 2006.

[66] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-binding: enabling unordered load-store queues. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 347–357, New York, NY, USA, 2007. ACM.

[67] R. Shillner. Clean: Removing useless control flow, June 1994.

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

[69] A. Smith, J. Burrill, R. McDonald, B. Yoder, D. Burger, S. W. Keckler, and K. S. McKinley. TRIPS application binary interface (ABI) manual. Technical Report TR-05-22, The University of Texas at Austin, May 2005. http://www.cs.utexas.edu/users/cart/trips.

[70] A. Smith, J. Gibson, J. Burrill, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. TRIPS intermediate language (TIL) manual. Technical Report TR-05-20, The University of Texas at Austin, May 2005. http://www.cs.utexas.edu/users/cart/trips.

[71] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[72] S. Swanson, K. Michaelson, and M. Oskin. Configuration by combustion: Online simulated annealing for dynamic hardware configuration, October 2002.

[73] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–302, December 2003.

[74] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report TR-370, LCS, MIT, Cambridge, MA, August 1986.

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

[76] F. Tseng. *Braids: out-of-order performance with almost in-order complexity*. PhD thesis, The University of Texas at Austin, 2007.

[77] F. von Leitner. diet libc - a libc optimized for small size.

[78] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 15–25, January 2001.

[79] N. J. Warter, D. M. Lavery, and W. W. Hwu. The benefit of predicated execution for software pipelining. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 497–506, January 1993.

[80] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 1993.

[81] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[82] B. Yoder, J. Burrill, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. TRIPS assembler and assembly language (TASL) specification. Technical Report TR-05-21, The University of Texas at Austin, May 2005. http://www.cs.utexas.edu/users/cart/trips.

[83] B. Yoder, J. Burrill, R. McDonald, K. Bush, K. Coons, M. Gebhart, S. Govindan, B. Maher, R. Nagarajan, B. Robatmili, K. Sankaralingam, S. Sharif, A. Smith, D. Burger, S. W. Keckler, and K. S. McKinley. Software infrastructure and tools for the trips prototype. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation*, June 2007.

[84] S. Zucker and K. Karhi. System V application binary interface: PowerPC processor supplement, September 1995.

Vita

Aaron Smith was born in Victoria, Texas on July 30th 1977, to Jesse Leal Smith and Pauline Mladenka Hendryx. He received a Bachelor of Arts degree in Computer Sciences in 1998, a Masters of Science in Computer Sciences in 2006, and a Bachelor of Arts in Japanese in 2008. All from the University of Texas at Austin. He is currently employed by Microsoft Research in Redmond, Washington.

Permanent address: 301 Woodlands Ln
Victoria, Texas 77904

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.