2016

Optimizing Transfers of Control in the Static Pipeline Architecture

Ryan R. Baird

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

OPTIMIZING TRANSFERS OF CONTROL IN THE STATIC PIPELINE ARCHITECTURE

By

RYAN R. BAIRD

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

2016

Ryan R. Baird defended this thesis on May 24, 2016.
The members of the supervisory committee were:

David Whalley

Professor Directing Thesis

Gary Tyson

Committee Member

Xin Yuan

Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Statically pipelined processors offer a new way to improve the performance beyond that of a traditional in-order pipeline while simultaneously reducing energy usage by enabling the compiler to control fine-grained details of the program execution. This paper describes how a compiler can exploit the features of the static pipeline architecture to apply optimizations on transfers of control that are not possible on a conventional architecture. The optimizations presented in this paper include hoisting the target address calculations for branches, jumps, and calls out of loops, performing branch chaining between calls and jumps, hoisting the setting of return addresses out of loops, and exploiting conditional calls and returns. The benefits of performing these transfer of control optimizations include a 6.8% reduction in execution time and a 3.6% decrease in estimated energy usage.

CHAPTER 1

INTRODUCTION

Power and energy have become critical design constraints in processors for several reasons. Some of these reasons include that mobile devices rely on low power usage to improve battery life, embedded devices have a limited power budget, processor clock rates are constrained due to thermal limitations, and electricity costs are increasing. Designing power efficient processors helps address all of these issues.

A recent approach to processor design, the Static Pipeline (SP), reduces energy usage by giving the compiler fine-grained control over the scheduling of pipeline effects. This approach enables the compiler to avoid many redundant pipeline actions, such as accesses to registers whose values are already available within the datapath. Additionally, this approach enables the processor to be simplified because design issues such as data forwarding and many hazards are handled by the compiler instead of the hardware.

The focus of this paper is to evaluate how a compiler can make transfers of control (ToCs) faster and more energy efficient in the SP architecture. The primary contributions of this paper are:

1. Adjusting ToCs in the SP architecture to deal with an extra stage in the pipeline
2. Providing a detailed description of how SP ToCs are implemented in the hardware
3. Implementing a variety of new SP ToC optimizations that exploit the decoupling of ToC effects in a fully datapath that would not otherwise be possible
4. evaluating the impact of these SP ToC optimizations.

The author has published the work outlined in this paper in [1].

1.1 Static Pipeline ISA

In order to illustrate the difference between a Static Pipeline ISA and a more traditional ISA, consider an example of a mips instruction: "lw $3, 4($5)". The programmer thinks of this in-

struction as a single operation (loading a value from memory), but it can be broken down into five operations that happen in the pipeline:

1. The retrieval of the immediate value 4 from the instruction
2. The read of register $5
3. The addition of the value 4 and the address from register $5
4. The load from memory based on the result from the addition
5. The write to register $3.

Typically, these operations happen in the Decode, Register Fetch, Execute, Memory, and Write-back stages of the pipeline; sequential operations are packaged into an instruction because we think of them as a single operation. At any given time, a traditional pipeline simultaneously processes one stage of several instructions. The processor is tasked with dynamically deciding when stalls have to be introduced due to hazards, and when values can be forwarded from different stages of the pipeline in order to avoid stalling.

The static pipeline tasks the compiler with determining when these operations are performed; instead leaving a sequence of effects packaged into an instruction, the compiler breaks the traditional instructions down into their micro-effect components, and schedules those micro-effects to determine which effects are executed at the same time within the processor. The effects that can be scheduled at the same time are combined into an instruction, and the scheduling of the effects must take into account limitations of the instruction encoding.

Figure 1.1 illustrates the difference between instruction effects in the SP approach compared to a traditional pipeline. The *load* instruction in Figure 1.1(b) requires one cycle for each stage after being fetched and decoded and remains in the pipeline from cycles six through nine. In a conventional processor, the sequence of effects composing the *load* is encoded in a single instruction but occurs over several cycles (in this case, 6-9). Figure 1.1(c) illustrates how an SP processor operates. Conventional operations, such as a load, still require multiple cycles to complete. However, the effects that make up a conventional operation are explicitely encoded, spread over multiple SP instructions. After decoding an instruction, an SP processor executes in parallel the effects encoded in the instruction, which originated from multiple conventional pipeline operations.

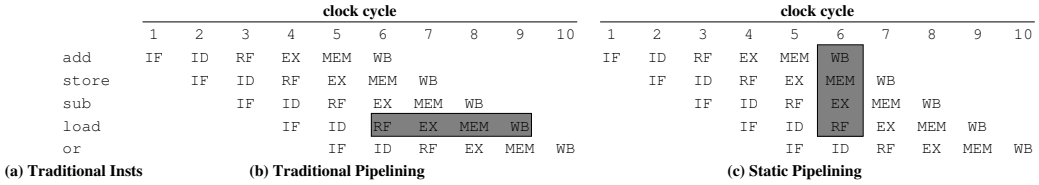

Figure 1.1: Traditionally Pipelined vs. Statically Pipelined Architecture

Unfortunately, allowing for every possible combination of effects in an SP instruction would require more than 80 bits per instruction. To keep the instruction size at 32 bits, we use a template based instruction encoding. In this encoding, we have several template forms, shown in Figure 1.2, which serve the purpose of keeping fields within templates aligned well enough to be easily decoded by the processor.

Each template is a set of effects that can be encoded if that template is selected. To generate templates, the compiler generates wide-format code, but restricts the instructions generated to be instructions that could possibly fit in a template. The template selector iteratively selects the template that can encode the most instructions which aren't covered by previously selected templates. The process for generating the templates is covered by more detail in [2].

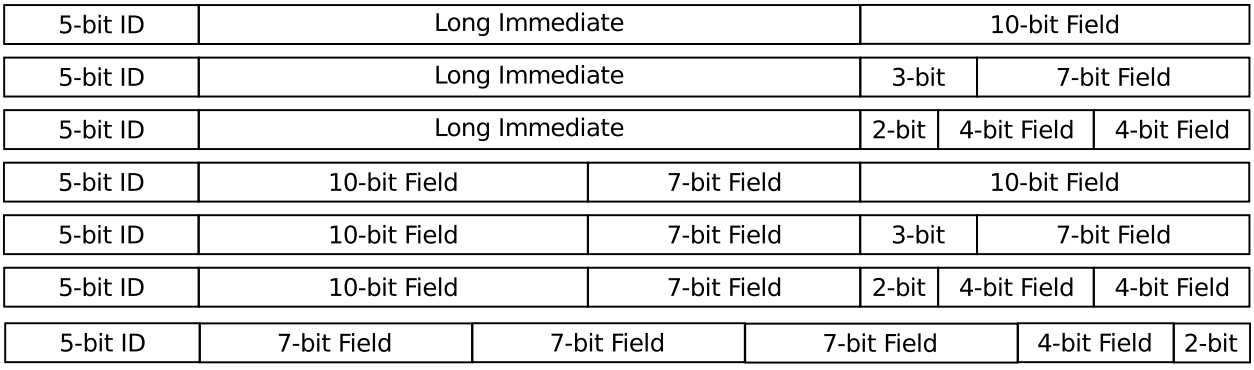

Figure 1.2: Static Pipeline Template Formats

1.2 Static Pipeline Architecture

A static pipeline's hardware is much simpler than a traditional pipeline. The processor fetches an instruction that represents a set of effects, decodes the instruction, and executes all of the effects in parallel. Since all of the effects have been scheduled ahead of time by the compiler, there isn't any need for forwarding logic and the hazard detection is much simpler.

A high-level overview of the datapath for our current SP design is shown in Figure 1.3. Most SP effects (all except stores, register writes and ToCs) update a dedicated internal register within the execute stage, which is cheaper than propagating information through the pipeline, storing it to a general purpose register file, and later retrieving it from the register file and propagating it back into the execute stage for use by another instruction (as done by a traditional processor). There are ten internal registers that can be accessed in SP instructions:

- The `SEQ` (sequential address) register gets the address of the next sequential instruction at the time it is written.
- The `RS1` and `RS2` (register source) registers contain source values read from the register file.
- The `SE` (sign extend) register contains a signed-extended immediate value.
- The `CP1` and `CP2` (copy) registers hold values copied from one of the other internal registers.
- The `OPER1` (ALU result) register receives values calculated in the ALU.
- The `OPER2` (FPU result) register acquires results calculated in the FPU, which is used for multi-cycle operations.
- The `ADDR` (address) register holds the result of an integer addition and is often used as an address to access either the instruction cache or data cache.
- The `LV` (load value) register gets assigned a value loaded from the data cache.

Each internal register requires less power to access than the centralized register file since these internal registers are small and can be placed near the portion of the processor that accesses them.

1.3 Changes to the Static Pipeline Architecture

The SP pipeline previously consisted of two stages, instruction fetch (IF) and execute (EX), in our simulations; instruction decode was assumed to be part of the EX stage. In order to make

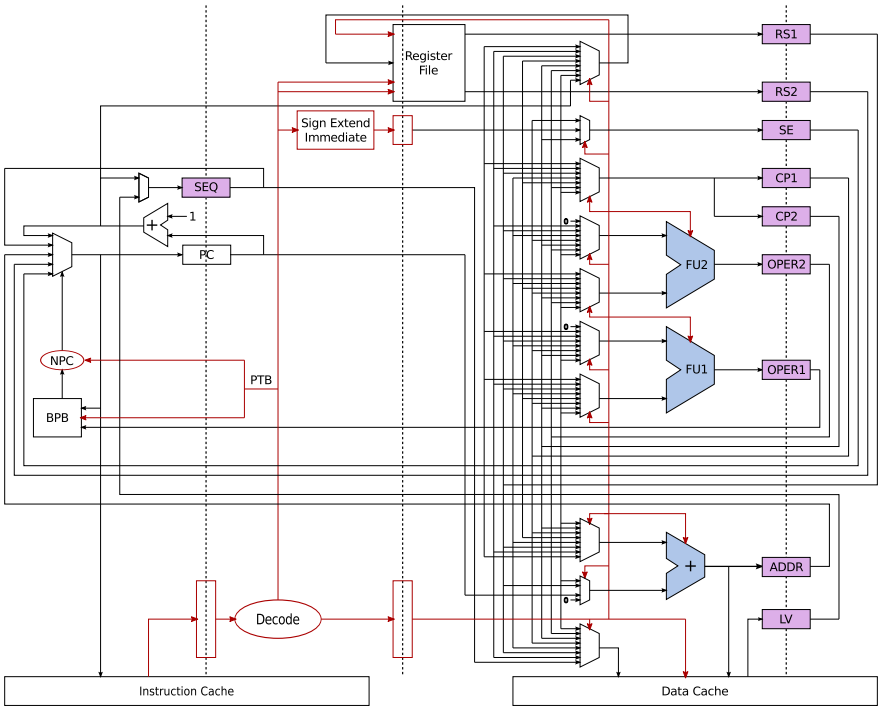

Figure 1.3: Static Pipeline Datapath

the clock period comparable to a baseline MIPS processor in a VHDL implementation for [2], we added an instruction decode (ID) stage between the IF and EX stages in which the instructions are converted into control signals. This change now requires that target addresses for ToCs be computed one instruction earlier. We also now require the address of a load/store operation to be calculated before the load/store effect is executed, which enables us to maintain 1-cycle loads for data cache accesses. The previous SP framework had a dedicated `TARG` register that was used to hold the address of a PC-relative target. We now use a single `ADDR` register for data memory address calculations, and for target address calculations for both conditional and unconditional PC-relative ToCs. The costs of these hardware changes are that all loads and stores go through a single adder (creating extra name dependencies), we sometimes require an extra effect to move an address to the `ADDR` register, there is an increase in dependence height for blocks ending with a ToC (creating extra instructions in small basic blocks), and branch mispredictions stall for an extra cycle.

CHAPTER 2

TRANSFERS OF CONTROL

In order to understand the benefits of the way our Static Pipeline handles transfers of control (ToCs), it's important to understand how ToCs are handled in both traditional pipeline architectures and static pipeline.

2.1 Traditional Pipeline Transfers of Control

Traditional pipelines often handle transfers of control by always performing extra work to remedy the fact that their design dictates a demand for information that isn't yet available. A common way of handling transfers of control is shown in Figure 2.1. In the IF stage during cycle 1, the processor accesses the BTB, BPB, and RAS. The BTB caches target addresses for ToC instructions; it can often recognize a branch that has already been seen and return the branch target address for direct jumps and conditional branches. The RAS keeps track of the return addresses on the call stack so that the innermost functions can return efficiently. The BPB is used to predict whether or not conditional branches will be taken. Since the processor does not know whether the instruction currently being fetched is a transfer of control, it accesses these structures during every cycle. The information from these structures determines what instruction will be fetched in cycle 2.

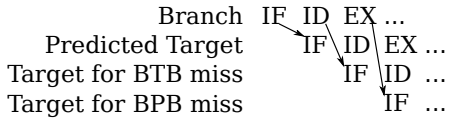

Figure 2.1: Traditional ToC

In the decode stage in cycle 2, the processor can determine whether or not the instruction fetched in cycle 1 was actually a ToC. If the instruction was a jump but was not in the BTB, the IF stage in cycle 2 needs to be cleared and the instruction fetched in cycle 3 will be the jump target.

During the execute stage in cycle 3, the condition can be evaluated for a branch fetched in cycle 1. If the prediction was incorrect, the fetch and decode stages need to be flushed and the correct instruction will be fetched in cycle 4 (correctly resolving the branch).

The processor is using extra energy to access predictive structures for every instruction that is not a branch, and to speculatively fetch an unnecessary instruction in the case of mispredictions.

2.2 Static Pipeline Transfers of Control

In an SP ToC, all of the information required to perform the ToC, with the exception of the branch condition, is always available before the point of the ToC. ToC operations (branches, jumps, calls, and returns) in an SP architecture are explicitly separated into three parts that span multiple SP instructions: (1) the target address calculation, (2) the prepare to branch (PTB) command, and (3) the point of the ToC.

Figure 2.2 depicts the way ToCs are handled in the Static Pipeline. The PTB and Address calculation provide the target address of the ToC before the instruction is fetched in cycle 2. The comparison occurs in cycle 3, and if a misprediction is revealed the IF and ID stage must be flushed and the correct instruction will be fetched in cycle 4. This approach still uses extra energy in the case of conditional branch mispredictions, but never makes accesses to the BPB for instructions that are not conditional branches, and doesn't require a BTB or an RAS.

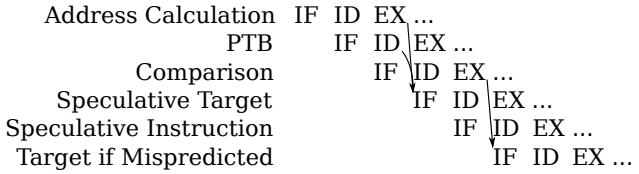

Figure 2.2: Static Pipeline ToC

Figure 2.3 provides examples of how ToCs are accomplished in the SP architecture. Before optimization, SP targets are always calculated by either adding the program counter (`PC`) and a constant for a PC-relative address or by using two long immediate effects to construct an absolute address, as depicted in Figures 2.3(a) and 2.3(b), respectively. After the target address is calculated, a prepare-to-branch (PTB) effect is issued. PTB instructions have been proposed in other architectures, but PTB effects have a low cost in the SP architecture because they can be encoded

as a 4-bit effect, rather than as an entire instruction. These 4 bits consist of one enable bit, one bit to select between conditional and unconditional ToCs, and 2 bits to select a source to obtain the target address. The possible SP target address sources are `ADDR` (PC-relative targets), `SE` values (direct call targets), `RS2` (originally used for only indirect targets such as returns), and `SEQ` (top of innermost loop targets). The PTB effect indicates that the point of the ToC is in the following instruction. If the conditional bit (`c`) is set, a comparison effect must be within the instruction immediately following the PTB. Calls and returns are made using the same control-flow effects as any other ToC. Instead of using a unified jump and link instruction for a call, we represent the set of the return address register as a separate effect to accomplish this goal (`r[31]=PC+1;`).

```
SE=offset(label);
ADDR=PC+SE;
...
PTB=c:ADDR;
PC=OPER1!=RS1,ADDR;
```

(a) PC-Relative SP Branch

```
SE=LO(label);
SE=SE|HI(label);
...
PTB=u:SE;
PC=SE;r[31]=PC+1;
```

(b) Absolute SP Call

Figure 2.3: SP Transfer of Control Examples

Figure 2.4 contains a pipeline diagram showing how the SP effects that comprise a ToC operation are pipelined. We require that the target address be assigned at least one instruction before the instruction containing the PTB effect is executed, as shown in Figure 2.4. Likewise, the PTB effect has to occur in the instruction immediately preceding the point of the ToC. The PTB effect is performed during the ID stage as it only determines whether or not the next instruction is a ToC, if the ToC is conditional or unconditional, and which source is used for the target address. In the diagram in Figure 2.4 both the target address calculation and the PTB effect are completed at the end of cycle 3. Thus, the exact target address is always known before the instruction at the target address is fetched, which occurs in cycle 4 in Figure 2.4.

One advantage of SP ToCs is that accesses to a branch target buffer (BTB) and a return address stack (RAS) are eliminated and many accesses to a branch prediction buffer (BPB) can be avoided. A conventional processor accesses a BTB, RAS, and a BPB on every cycle. The BTB and RAS are accessed in the IF stage in a conventional processor because the processor otherwise does not have the target address early enough to perform a branch in the decode stage, and are

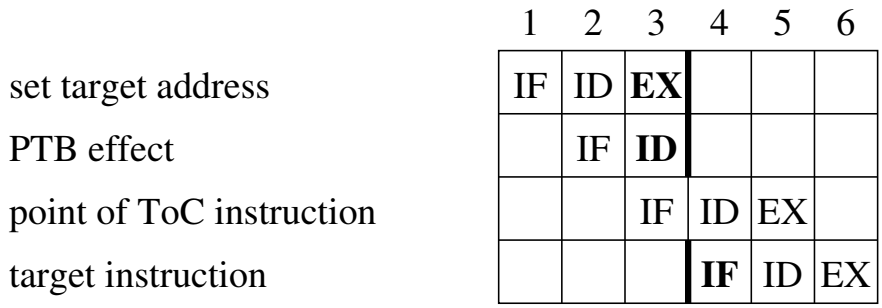

Figure 2.4: Pipelining SP Transfers of Control

accessed for every instruction because the processor does not differentiate between different types of instructions until the decode stage. Since our SP processor's PTB effect specifies the target address and that the next instruction is a ToC, our SP processor has no need for a BTB or RAS; both structures are completely removed. Removing the need for a BTB has a significant impact on energy usage since the BTB is a large and expensive structure to always access during the IF stage. A branch prediction buffer (BPB) in a conventional processor is also accessed in the IF stage for every instruction and contains bits to indicate if the branch is predicted to be taken or not taken. Since conditional ToCs in an SP processor are indicated in the PTB effect that immediately precedes the point of the ToC, the SP processor only needs to access the BPB for conditional branches.

There are several other advantages of breaking a ToC operation into separate effects that occur in different instructions. Most ToCs are to direct targets, meaning that the target address does not change during the application's execution. One advantage of decoupling these effects is that the compiler can perform transformations on the target address calculation that are not possible using a conventional instruction set where these calculations are tightly coupled with ToC instructions. For instance, by decoupling the target address calculation from the point of the ToC, the calculation can be hoisted out of loops. Likewise, the target address calculation for multiple ToCs to the same target address can be done once. Thus, many redundant target address calculations can be eliminated.

Other SP ToC optimizations are possible. Since unconditional jump, call, and return operations are also separated into three parts, the compiler can perform additional optimizations, such as chaining between jumps and calls and hoisting return address assignments out of loops. The SP

processor supports both direct and indirect sources for conditional and unconditional ToCs, which the compiler can exploit by converting conditional branches that are preceded by a call or followed by a call or a return into conditional calls and conditional returns. Conditional indirect jumps are not available in most conventional instruction sets.

2.3 Compilation for the SP Architecture

In this section, we describe the overall compilation process in more detail. For an SP architecture, the compiler is responsible for controlling each portion of the datapath during each cycle, so effective compiler optimizations are critical to achieve acceptable performance and code size. Because the instruction set architecture (ISA) for an SP processor is quite different from that of a RISC architecture, many compilation strategies and optimizations have to be reconsidered when applied to an SP architecture.

Figure 2.5 shows the steps of our compilation process. First, C code is input to the *frontend*, which consists of the LCC compiler [3] frontend that converts LCC's output format into the register transfer list (RTL) format used by the VPO compiler [4].

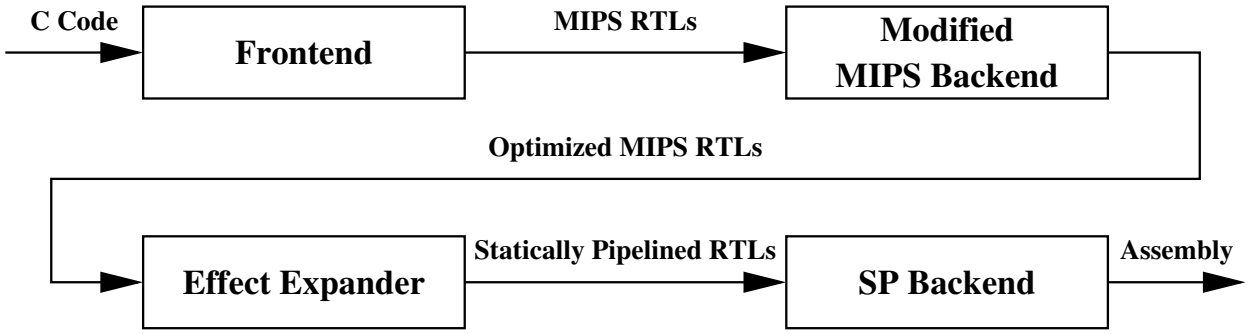

Figure 2.5: SP Compilation Process

These RTLs are then input into a *modified MIPS backend*, which performs all the conventional compiler optimizations applied in VPO with the exception of instruction scheduling. These optimizations are performed before conversion to SP instructions because many are more difficult to apply on the lower level SP representation, which breaks many assumptions in a conventional compiler backend. VPO's optimizations include those typically performed on ToCs, such as branch chaining, reversing branches to eliminate unconditional jumps, minimizing loop jumps by dupli-

cating a portion of a loop, reordering basic blocks to eliminate unconditional jumps, and removing useless conditional branches and unconditional jumps whose target is the following positional block. This strategy enables us to concentrate on optimizations specific to the SP as all conventional optimizations have already been performed.

The *effect expander* breaks the MIPS instructions into instructions that are legal for the SP. This process works by expanding each MIPS RTL into a sequence of SP RTLs, each containing a single effect, that together perform the same computation. Thus, ToCs are also broken into multiple effects at this point.

Lastly, these instructions are fed into the *SP backend*, also based on VPO, which was ported to the SP architecture since its RTL intermediate representation is at the level of machine instructions. A machine-level representation is needed for performing code improving transformations on SP generated code. This backend applies additional optimizations, which include the SP ToC optimizations described in this paper, and produces the final assembly code.

CHAPTER 3

SP TRANSFER OF CONTROL OPTIMIZATIONS

Several new ToC optimizations are possible and beneficial due to the way ToCs are represented in the SP architecture. The relevant optimizations we describe are (1) using the SEQ register to hoist top of innermost loop target address calculations out of loops, (2) using general-purpose registers to hoist other target address calculations out of loops, (3) performing call-jump and jump-call chaining, (4) hoisting return address assignments out of loops, and (5) exploiting conditional calls and returns. With the exception of the SEQ hoisting optimization, all of these optimizations are new as compared to prior SP studies [5, 6].

3.1 Hoisting the Top of Innermost Loop Target

In a conventional ISA, address calculations are typically tightly coupled with ToC instructions; even though the target addresses do not change, they are recalculated every loop iteration. For PC-relative ToCs requiring addition, this means the addition is performed every time the ToC is encountered, which requires both additional energy usage and encoding space. Even for ToCs that use an absolute PC address, the encoding space for that address is still required in the ToC instruction. The encoding space for calculating SP target addresses could impact performance if not hoisted out of loops since more instructions may need to be executed.

The most frequent ToCs in an application are typically in the innermost loops of the most deeply nested functions. The SP architecture provides the `SEQ` internal register which can be set to the next sequential instruction address. The operations involving the `SEQ` register that are supported by the SP architecture are (1) assigning the incremented value of the program counter to the `SEQ` register (`SEQ=PC+1;`), (2) storing the `SEQ` register value to memory (`M[ADDR]=SEQ;`), and (3) assigning the value from the `LV` register (result of a load operation) to the `SEQ` register (`SEQ=LV;`). The last two operations are used to save and restore the `SEQ` register value so that its value can be preserved across a function call. The compiler exploits this register by assigning it

the address of the top-most instruction of an innermost loop, which allows the elimination of any calculations of this address within the body of the loop.

The example in Figure 3.1 depicts the RTLs within an innermost loop that contains ToCs to the top of a loop. All of the examples in this paper showing SP ToC optimizations are depicted at the time the optimizations are applied, which is before multiple effects are scheduled in each instruction. PTB effects are actually inserted during scheduling, but are included to clarify the examples. The compiler exploits the `SEQ` register by placing the `SEQ=PC+1;` effect in the last instruction of the block immediately preceding the innermost loop. This block has to dominate the header of the loop, which is usually the case as the block is typically the loop preheader. Thus, executing this effect results in the address of the top-most instruction in the loop (`L1` in Figure 3.1) being assigned to the `SEQ` register. The top-most block in the loop is not always the loop header, but is always a target of one or more ToCs within the loop. The compiler then modifies all conditional and unconditional ToCs to the top-most block of the loop to reference the `SEQ` register instead of performing a PC-relative address calculation. Two effects in the loop are eliminated for each ToC referencing the `SEQ` register, which on average improves performance because it often decreases the number of instructions in the loop after scheduling SP effects. For instance, Figure 3.1(a) has two conditional ToCs in the loop that both have the same target, which is the top-most instruction within the loop and both of the ToCs can now just reference the `SEQ` register after the transformation, as depicted in Figure 3.1(b).

```
for(...)  {
   if(...)  {
      ...
   }
}
```

(a) Loop at Source Code Level

```
L1 # Beginning of loop
   ...
   SE=offset(L1);
   ADDR=PC+SE;
   PTB=c:ADDR;
   PC=RS2==LV,ADDR(L1);
   ...
   SE=offset(L1);
   ADDR=PC+SE;
   PTB=c:ADDR;
   PC=OPER1!=CP1,ADDR(L1);
```

(b) Before SEQ Optimization

```
   SEQ=PC+1;
L1 # Beginning of loop
   ...
   PTB=c:SEQ;
   PC=RS2==LV,SEQ(L1);
   ...
   PTB=c:SEQ;
   PC=OPER1!CP1,SEQ(L1);
```

(c) After SEQ Optimization

Figure 3.1: Example of SEQ Optimization

3.2 Hoisting Other Target Address Calculations

There are often several other ToCs in loops whose targets are not to the top-most instruction in an innermost loop. In a prior version of the SP datapath [6], only one other target address calculation was hoisted out of loops into a dedicated internal `TARG` register, which is no longer supported in the SP datapath. The compiler now hoists these target address calculations out of the loop using an integer register when one is available, which can reduce both execution time (because there are fewer effects to schedule) and energy usage.

We hoist both PC-relative target address calculations (conditional branches and unconditional jumps) and absolute target address calculations (direct calls) out of loops using registers from the integer register file. Due to the irregularity of the SP ISA, conventional loop-invariant code motion is unable to hoist these target address calculations. The algorithm for this optimization, which is shown in Figure 3.2, hoists target address calculations starting with the innermost loops. When there are multiple target address calculations in a given loop, the compiler must prioritize which ones to hoist as each target requires a separate register and there are a limited number of available registers. The prioritization is based on estimated benefits. We consider the likelihood of the block containing the ToC being executed to be the most important factor as hoisting a computation that rarely gets executed would not be beneficial. The next factor is the number of ToCs in the loop to the same target, as a single register assigned the target address outside the loop can replace multiple target address calculations inside the loop. The next factor is if an absolute target address calculation is performed versus a PC-relative target address calculation. An absolute target address calculation occurs for direct calls and requires two long immediate effects, which each require 17 bits (see Figure 1.2). In contrast, a PC-relative target address calculation is used for conditional branches and unconditional jumps and typically requires a short immediate and an integer addition, which each require 7 bits (see Figure 1.2). We consider the least important factor to be the number of instructions in the basic block containing the ToC. The effects associated with an absolute or PC-relative target address calculation do not have any true dependences with other effects in the loop. A basic block with fewer instructions will likely have fewer available slots to schedule the target address calculation effects with the instructions comprising the other SP effects within the block. A target address calculation can only be hoisted out of a loop if an integer register is available

to hold the target address and RS2 is available at the point of the ToC since RS2 is used to read the register value and serve as the target specified in a PTB effect.

```
FOR each loop L (innermost first) DO
    create list A of all targets of ToCs at outer level in L
    prioritize the order of A based on following constraints:
         1.  estimated frequency of block containing target calc
         2.  number of target calcs to that target
         3.  absolute over PC-relative target calcs
         4.  fewest instructions in block containing target calc
    WHILE a register R is available DO
         IF RS2 available at ToCs in first target in A THEN
            assign to T first target in A not yet hoisted
            place target calc C for T in L's preheader
            after C assign C's result to R
            replace target calc(s) of T in L with reads of R
```

Figure 3.2: Algorithm for Hoisting Target Address Calculations

Figure 3.3 depicts an example of applying this optimization within a loop nest. Figure 3.3(a) shows C source code that results in five ToCs in SP instructions, which are two conditional branches associated with the `if` statement, one direct call associated with the call to `f`, and two conditional branches associated with the `for` statements. Figure 3.3(b) shows the SP instructions, where the conditional branch associated with the inner `for` statement already has a target of `SEQ`. The two conditional branches associated with the `if` statement both have `L2` as a PC-relative target address. Having multiple branches to the same target is common when logical AND or OR operators are used in conditional expressions. The call to `f` is constructed using two large immediate effects. Figure 3.3(c) shows the SP instructions after applying this optimization.

The target address calculation of `L2` and `f` have been hoisted out of the loop nest and their values have been stored in `r[17]` and `r[18]`, respectively. Storing the address of `PC+1` into a register was designed to be used to store the return address into `r[31]` for a call, but we now use this effect to assign `PC+1` to a register to the address of the topmost block of outer loops (`r[19]=PC+1;`). Each modified ToC in the loop now requires three effects instead of four. The target address calculations associated with the ToCs have been replaced with the appropriate register read effect. Note that only one target address calculation is performed for `L2`, where two distinct calculations are required

```
for (...) {
    for (...)  {
        ...
        if (...&&...)
            f();
        ...
    }
}
```

(a) Loop at Source Code Level

```
L1 # start of outer loop
   ...
   SEQ=PC+1;    # SEQ=L2
L2 # start of inner loop
   ...
   SE=offset(L3);
   ADDR=PC+SE;  # 1st if
   PTB=c:ADDR;  # ToC
   PC=OPER1!=RS1,ADDR(L3);
   ...
   SE=offset(L3);
   ADDR=PC+SE;  # 2nd if
   PTB=c:ADDR;  # ToC
   PC=LV!=RS1,ADDR(L3);
   ...
   SE=LO:f;     # call to
   SE=SE|HI:f;  # f
   PTB=u:SE;
   PC=SE(f);r[31]=PC+1;
L3 ...          # inner for
   PTB=b:SEQ;   # ToC
   PC=OPER1!=SE,SEQ(L2);
   ...
   SE=offset(L1);
   ADDR=PC+SE;  # outer for
   PTB=b:ADDR;  # ToC
   PC=OPER1!=SE,ADDR(L1);
```

(b) Loop after SEQ Transformation

```
   SE=offset(L3);
   ADDR=PC+SE;
   r[17]=ADDR;   # r17=L3
   SE=LO:f;
   SE=SE|HI:f;
   r[18]=SE;     # r18=f
   r[19]=PC+1;   # r19=L1
L1 # start of outer loop
   ...
   SEQ=PC+1;     # SEQ=L2
L2 # start of inner loop
   ...
   RS2=r[17];    # 1st if
   PTB=c:RS2;    # ToC
   PC=OPER1!=RS1,RS2(L3);
   ...
   RS2=r[17];    # 2nd if
   PTB=c:RS2;    # ToC
   PC=LV!=RS1,RS2(L3);
   ...
   RS2=r[18];    # call to
   PTB=u:RS2;    # f
   PC=RS2(f);r[31]=PC+1;
L3 ...           # inner for
   PTB=b:SEQ;    # ToC
   PC=OPER1!=SE,SEQ(L2);
   ...
   RS2=r[19];    # outer for
   PTB=b:RS2;    # ToC
   PC=OPER1!=SE,RS2(L1);
```

(c) Loop after Hoisting other Target Address Calculations

Figure 3.3: Example of Target Address Calculation Hoisting

in the loop in Figure 3.3(b) if the values of `SE` or `ADDR` are overwritten between the two ToCs. The second read of `r[17]` in Figure 3.3(c) will be eliminated after common subexpression elimination is applied if `RS2` is not overwritten before that point. The transformation increases the static number of effects and often the overall code size, but decreases the dynamic number of effects and often

the number of instructions within a loop, which results in improvements in energy usage and often performance.

3.3 Performing Call-Jump and Jump-Call Chaining

When a call is followed by an unconditional jump or the target block of an unconditional jump contains a call, the unconditional jump can be eliminated by adjusting the return address.

Figure 3.4 shows how a jump is eliminated when a call is followed by a jump. We move the instructions between the call and the jump to before the call when there are no dependences so this transformation can be more frequently applied. Figure 3.4(a) shows a call to `f1` in the `then` portion of an `if-then-else` statement. As shown in Figure 3.4(b), the call to `f1` is followed by an unconditional jump to `L2` that jumps over the `else` portion of the `if-then-else` statement. Figure 3.4(c) shows that the jump to `L2` is eliminated and before the call `r[31]` is now assigned the address of `L2`, which was the target of the unconditional jump. Besides eliminating the PTB and PC effects of the unconditional jump, this transformation also places the target address calculation of the jump target at a point where it can be scheduled in parallel with effects preceding the call and effects associated with the call itself.

```
if (...) {
   ...
   f1();
   ...
}
else {
   ...
}
```

(a) Call Followed by Jump at Source Code Level

```
SE=LO:f1;
SE=SE|HI:f1; # call
PTB=u:SE;    # to f1
PC=SE(f1);r[31]=PC+1;
...
SE=offset(L2);
ADDR=PC+SE;  # jump
PTB=u:ADDR;  # over
PC=ADDR(L2); # else
```

(b) Before Chaining Call to Jump

```
...
SE=offset(L2);
ADDR=PC+SE;
r[31]=ADDR;  # call
SE=LO:f1;    # to f1
SE=SE|HI:f1; # with
PTB=u:SE;    # return
PC=SE(f1);   # to L2
```

(c) After Chaining Call to Jump

Figure 3.4: Example of Call-Jump Chaining

Figure 3.5 shows how a jump is eliminated when a jump is followed by a call. Figure 3.5(a) shows a call to `f2` after an `if-then-else` statement. As shown in Figure 3.5(b), the target block, `L4`, of

the unconditional jump contains a call to `f2`. Figure 3.5(c) shows that the call to `f2` is duplicated at the point of the unconditional jump, the jump to `L4` is eliminated, and `r[31]` is now assigned the address of `L5`, which is the address of the instruction following the call. The transformation eliminates two effects (assignments to PTB and PC) at the expense of duplicating the call. Jump-call chaining is more aggressively performed than call-jump chaining since instructions at the jump target preceding the call can always be duplicated in the jump block.

```
if (...) {
   ...
}
else {
   ...
}
...
f2();
```

(a) Jump Followed by Call at Source Code Level

```
      SE=offset(L4);
      ADDR=PC+SE;  # jump
      PTB=u:ADDR;  # over
      PC=ADDR(L4); # else
      ...
L4    ...
      SE=LO:f2;    # call
      SE=SE|HI:f2; # to f2
      PTB=u:SE;
      PC=SE(f2);r[31]=PC+1;
      ...
```

(b) Before Chaining Jump to Call

```
      ...
      SE=offset(L5);
      ADDR=PC+SE;
      r[31]=ADDR;  # call to
      SE=LO:f2;    #    f2
      SE=SE|HI:f2; #    with
      PTB=u:SE;    #    return
      PC=SE(f2);   #    to L5
      ...
L4    ...
      SE=LO:f2;    # call to
      SE=SE|HI:f2; #    f2
      PTB=u:SE;
      PC=SE(f2);r[31]=PC+1;
L5    ...
```

(c) After Chaining Jump to Call

Figure 3.5: Example of Jump-Call Chaining

An interesting note about these call-jump and jump-call chaining optimizations is that both could be performed in a conventional ISA by updating the return address register and using a jump instead of a call instruction. However, such an optimization would not be beneficial for any processor with a return address stack (RAS) because the resulting code would perform more return address pops than pushes, which would result in returns to the wrong address. Note that the SP architecture eliminates the need for a RAS since the return address is known at the point of the return ToC.

3.4 Hoisting Return Address Assignments

In SP generated code, the return address is set in an effect that is independent from the PTB effect causing the ToC associated with the call operation. In some cases, it can be beneficial to set the return address outside of the loop. Since the return address register is callee-save, loops with a single call or loops for which all calls can be made to return to the same instruction do not need to set the return address register every loop iteration.

Figure 3.6 shows the algorithm for hoisting return address assignments out of loops. The optimization we implemented examines each call within a loop, starting with the outermost loop as it can hoist at most one return address assignment out of a loop nest since the return address has to be assigned to the single return address register `r[31]`. For each call in the loop, the compiler determines the instruction associated with the return address. If the call is immediately followed by an unconditional jump, then the return address is associated with the target of the unconditional jump. This requires skipping over any address calculations and checking if all values computed before the jump are dead at the point of the jump. If the return address differs for any two calls or if no registers are available, then the optimization is not performed. Otherwise, the return address assignment is placed in the preheader of the loop and the return address assignments within the loop are removed along with any jumps to the common return target that follow a call.

Figure 3.7 depicts an example of this transformation performed on multiple calls within a single loop. Figure 3.7(a) shows a loop with calls to functions `f1` and `f2`. The ToCs in the loop include one for the `if` statement condition, one for each call, one for the unconditional jump at the end of the `then` portion of the `if` statement, and one for the `for` statement condition. Figure 3.7(b) shows the SP code after hoisting all the target address calculations out of the loop. At this point there are assignments to `r[31]` in the loop at each call. Note that the call to `f1` is followed by an unconditional jump to `L3` and the return address from the call to `f2` is also `L3`. Figure 3.7(c) shows the SP code after hoisting the two return address assignments to `r[31]` out of the loop. The instructions comprising the unconditional jump after the call to `f1` are eliminated since these instructions can no longer be reached in the control flow. Likewise, the target address calculation instructions resulting in the assignment to `r[20]` in the loop preheader are eliminated since these assignments are now dead after the removal of the unconditional jump. In this example, the return

```
FOR each loop L (outermost first) DO
    IF a register unavailable within L THEN
         CONTINUE
    create list C of all calls in L
    FOR each C DO
         IF instructions following C comprise a
            direct unconditional jump J THEN
            associate return address of C to be target of jump J
         ELSE
            associate return address of C to be instruction after C
    IF any two C's have different return addresses THEN
         CONTINUE
    FOR each C DO
         IF C is followed by a direct jump J THEN
            remove instructions comprising jump J
         remove return address assignment of C
    place return address assignment to common target in preheader
    BREAK
```

Figure 3.6: Hoisting Return Address Assignments Algorithm

address hoisting transformation reduces the overall code size, the number of ToCs executed, and the energy usage required to execute the code.

3.5 Exploiting Conditional Calls and Returns

The SP ISA enables PC-relative, absolute, and indirect addresses to be used for both conditional and unconditional ToCs. We exploit these features in our compiler by introducing conditional calls and conditional returns without any changes to the SP architecture. A conditional branch where one successor goes directly to a call or return can in some circumstances be replaced with a conditional branch directly to the call target or return address. If a conditional branch falls into the call or return, then the condition must be reversed. If a conditional branch falls into a call, then the first instruction after the call must be the target of the original branch.

We don't perform conditional returns in functions that will have a stack, and we don't perform conditional calls in any case where arguments would have to be loaded into registers speculatively. If there are a couple of effects calculating or loading the address for a call, we speculatively compute

```
for (...) {
   ...
   if (...) {
      ...
      f1();
   }
   else {
      ...
      f2();
   }
   ...
}
```

(a) Loop with Calls at Source Code Level

```
      SE=offset(L3);
      ADDR=PC+SE;
      r[20]=ADDR; # r20=L3
      SEQ=PC+1;   # SEQ=L1
   L1 # Beginning of loop
      ...
      RS2=r[17];  # if stmt
      PTB=c:RS2;  # ToC
      PC=OPER1!=RS1,RS2(L2);
      ...
      RS2=r[18];  # call to
      PTB=u:RS2;  # to f1
      PC=RS2(f1);r[31]=PC+1;
      RS2=r[20];
      PTB=u:RS2;  # jump
      PC=RS2(L3); # to L3
   L2 ...
      RS2=r[19];  # call
      PTB=u:RS2;  # to f2
      PC=RS2(f2);r[31]=PC+1;
   L3 ...         # for stmt
      PTB=b:SEQ;  # ToC
      PC=OPER1!=SE,SEQ(L1);
```

(b) Loop without Return Address Assignment Hoisting

```
      SE=offset(L3);
      ADDR=PC+SE;
      r[31]=ADDR; # r31=L3
      SEQ=PC+1;   # SEQ=L1
   L1 # Beginning of loop
      ...
      RS2=r[17];  # if stmt
      PTB=c:RS2;  #    ToC
      PC=OPER1!=RS1,RS2(L2);
      ...
      RS2=r[18];  # call to
      PTB=u:RS2;  #    f1
      PC=RS2(f1);
   L2 ...
      RS2=r[19];  # call to
      PTB=u:RS2;  #    f2
      PC=RS2(f2);
   L3 ...         # for stmt
      PTB=b:SEQ;  #    ToC
      PC=OPER1!=SE,SEQ(L1);
```

(c) Loop with Return Address Assignment Hoisting

Figure 3.7: Example of Return Address Assignment Hoisting

the address in order to produce a conditional call. Similarly, we speculatively load the return address into RS2 for a return, because this is almost always required.

Figure 3.8 depicts an example of exploiting a conditional return. Figure 3.8(a) shows a source code fragment and Figure 3.8(b) shows the corresponding SP instructions. This transformation can only be applied when the return immediately follows the taken path (`L4`) of the branch, meaning the current function must be a leaf and no space is used for an activation record (no adjustment of the stack pointer and no restores of register values). Figure 3.8(c) shows the SP instructions after performing the optimization. The branch target is set to the return address and the original address calculation is removed by dead assignment elimination. Note the second read of `r[31]` will be eliminated if there is no assignment to `RS2` between the conditional return and the return.

Branches to calls are handled in a similar manner.

```
if (...) {
   ...
}
return;
```

(a) Branch Followed by Return at Source Code Level

```
   ...
   SE=offset(L4);
   TARG=PC+SE; # if stmt
   PTB=c:ADDR; #    ToC
   PC=LV==SE,ADDR(L4);
   ...
L4 RS2=r[31];  # return
   PTB=u:RS2;
   PC=RS2;
```

(b) Without Conditional Return

```
   ...
   RS2=r[31]; # cond
   PTB=c:RS2; # return
   PC=LV==SE,RS2;
   ...
L4 RS2=r[31];  # return
   PTB=u:RS2;
   PC=RS2;
```

(c) With Conditional Return

Figure 3.8: Example of Return Address Hoisting with Conditional Return

Exploiting conditional calls between a branch and its successor requires not changing the behavior or adversely affecting the performance when the branch has a different outcome. We found that we can exploit conditional calls more frequently when a call precedes a conditional branch. The requirements are that a call precedes a branch and the effects between the call and the branch can be moved before the call. Figure 3.9 depicts an example of exploiting a conditional call. Figure 3.9(a) shows the source code of a loop and assume `i` and `n` are local variables that are not affected by the call to `f`. A loop branch often just controls the number of times the loop iterates and is independent of a preceding call. Figure 3.9(b) shows the corresponding SP instructions. The call precedes the branch, and the address of `f` and the return address `L3` assignment have been hoisted out of the loop. Figure 3.9(c) shows the SP instructions after performing the optimization. The effects after the call have been moved before the call, the address of the branch target `L2` has been stored in `r[31]` in the preheader, the target of the branch is now the address of the called function, and the original call is moved after the branch. The called function from the loop will directly return to the original branch target `L2`. The call after the loop is needed since the call in the last original loop iteration still has to occur when the branch is not taken.

```
for (i=0; i<n; i++) {
   ...
   f();
}
```

(a) Call at End of a Loop at the Source Code Level

```
   SE=LO:f;
   SE=SE|HI:f;
   r[17]=SE;   # r17=f
   SE=offset(L3);
   ADDR=PC+SE;
   r[31]=ADDR; # r31=L3
   SEQ=PC+1;   # SEQ=L2
L2 # Beginning of loop
   ...
   RS2=r[17];  # call to
   PTB=u:RS2;  #    f
   PC=RS2(f);
L3 ...
   PTB=c:SEQ;
   PC=OPER1!=CP2,SEQ(L2);
```

(b) Without a Conditional Call

```
   SE=LO:f;
   SE=SE|HI:f;
   r[17]=SE;   # r17=f
   r[31]=PC+1; # r31=L2
L2 # Beginning of loop
   ...
L3 ...
   RS2=r[17];  # cond call
   PTB=c:RS2;  #    to f
   PC=OPER1!=CP2,RS2(f);
   PTB=u:RS2;  # call to f
   PC=RS2(f);r[31]=PC+1;
```

(c) With a Conditional Call

Figure 3.9: Example of Return Address Hoisting with a Conditional Call

CHAPTER 4

EVALUATION

In this section we describe the experimental environment and results from applying SP ToC optimizations.

4.1 Experimental Setup

We run the same 17 benchmarks from the MiBench benchmark suite [7] as all previous static pipeline publications, shown in Table 4.1, which is representative of embedded applications. To compile our baseline, we ran the VPO MIPS port with all optimizations enabled. We compiled benchmarks for the Static Pipeline processor using the process covered in Section 2.3. For each architecture we run a GNU assembler and linker to generate a MIPS executable. We used a simulator based on the SimpleScalar in-order MIPS simulator [8] to simulate the benchmarks.

We didn't compile the standard library; library calls were simulated in MIPS for both benchmarks. For the SP simulations, more than 90% of the dynamic instruction count consisted of Static Pipeline instructions; the others were in standard library routines. All cycles and structure accesses we reference for power are always counted toward the results regardless of whether they come from the MIPS libraries or our compiled SP code.

Table 4.1: Benchmarks Used

Category	Benchmarks
automotive	bitcount, qsort, susan
consumer	jpeg, tiff
network	dijkstra, patricia
office	ispell, stringsearch
security	blowfish, rijndael, pgp, sha
telecom	adpcm, CRC32, FFT, GSM

The simulator we used has been extended to include a bimodal branch predictor with 256 two-bit saturating counters, and a 256 entry branch target buffer (BTB) used when simulating MIPS

instructions. The simulator has also been extended to include level one data and instruction caches; each of these caches has 256 direct-mapped 32-byte lines of data.

Each of the graphs in the following sections represent the ratio between SP code to MIPS code. A ratio less than 1.0 means that the SP has reduced the value, while a ratio over 1.0 means that the SP has increased the value. When a given benchmark had more than one simulation associated with it (e.g., *jpeg* has both encode and decode), we averaged all of its simulations to avoid weighing benchmarks with multiple runs more heavily.

To estimate the energy savings, we counted events such as register file accesses, branch predictions, cache accesses, and ALU operations. Get a useful energy estimate out of this data requires an estimate of how much power each of these events cosumes relative to one another. We used estimates that were generated by using CACTI [9] to model the SRAMS within the pipeline, and by synthesizing other components for a 65nm processor. The components were simulated at the netlist level to determine the average case activation power, which was then normalized to the power of a 32-entry dual-ported register file read. The ratios between component power are dependent on processor technology, but the difference should not large enough to have a big impact on the final estimated ratios. The total energy estimate for a simulation is given by the weighted sum of the component accesses, where the weight is equal to the relative access power for each component, which is given in Table 4.2.

Table 4.2: Pipeline Component Relative Power

Component	Relative Access Power
Level 1 Caches (8kB)	5.10
Branch Prediction Buffer	0.65
Branch Target Buffer	2.86
Register File Access	1.00
Arithmetic Logic Unit	4.11
Floating Point Unit	12.60
Internal Register Writes	0.10

4.2 Results

There were more PC-relative calculations in the SP code than the MIPS code before our optimizations, because calculations are sometimes speculatively performed due to cross block schedul-

ing, unconditional jumps on the MIPS were performed using absolute addresses, and SP unconditional jumps used PC-relative target addresses. The number of PC-relative target address calculations improved from a ratio of 1.04 to a ratio of 0.44, as shown in Figure 4.1. This improvement primarily came from utilization of the `SEQ` register and use of integer registers to hoist target address calculations out of loops.

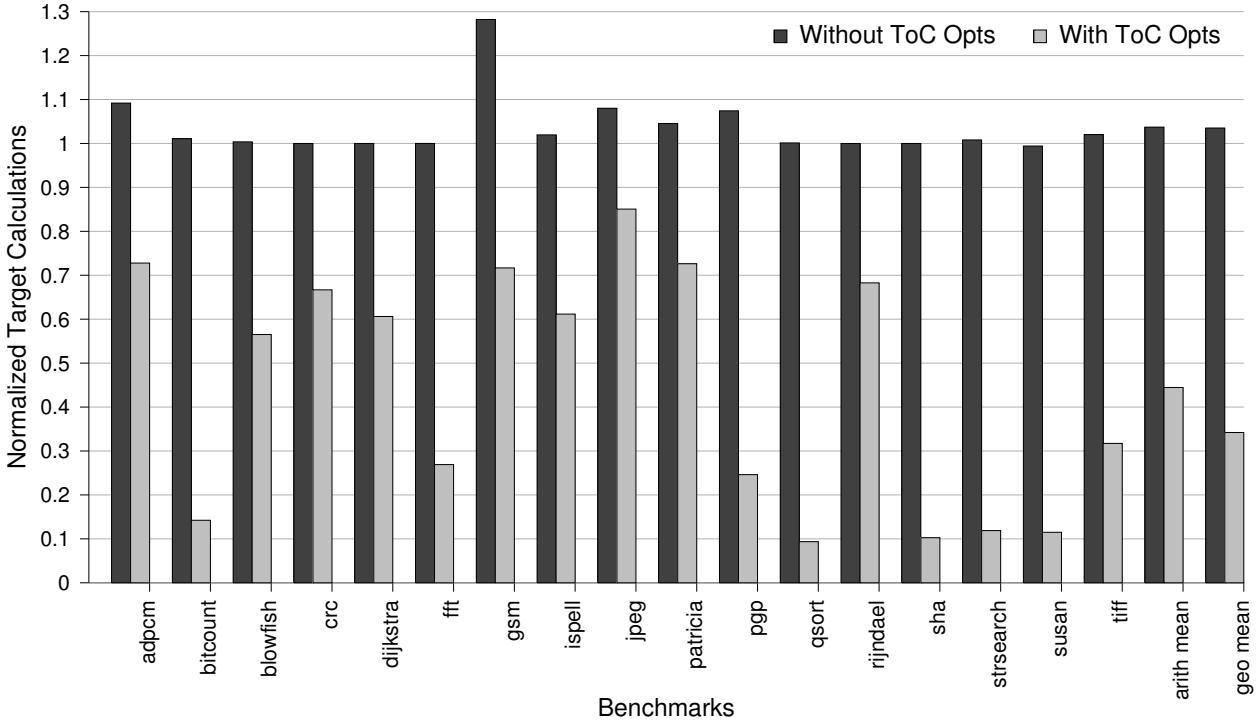

Figure 4.1: PC-Relative Target Address Calculation Ratio

The number of absolute target address calculations improved from a ratio of 1.00 to a ratio of 0.83, as shown in Figure 4.2. This improvement came from performing fewer target address calculations of calls inside loops. This improvement is less than the improvement for PC-relative address calculations, because direct calls are not always in loops, but every loop in the benchmarks is expected to have at least one conditional branch. Also, the presence of a call in a loop means that only the callee-save registers are available for hoisting target address calculations. Sometimes the number of absolute target address calculations increases after hoisting the calculation of out a loop when the direct call is rarely executed in the loop due to conditional control flow.

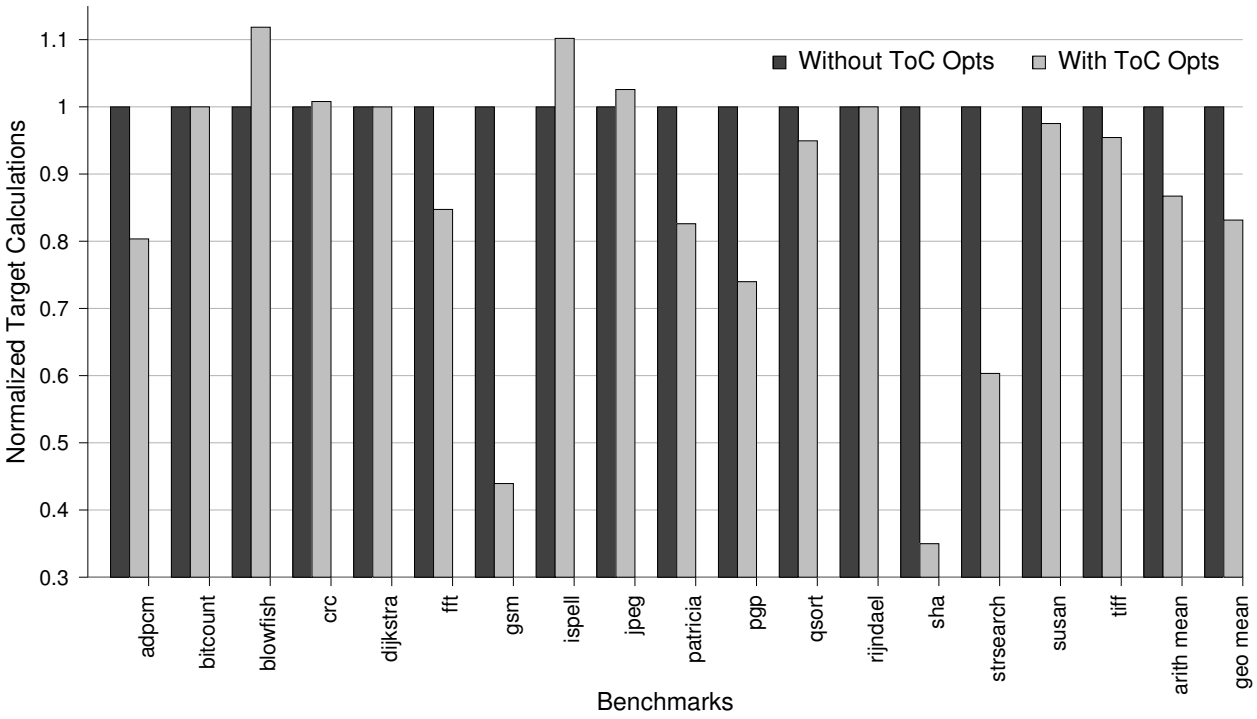

Figure 4.2: Absolute Target Address Calculation Ratio

On average, we were able to reduce the number of return address assignments to a ratio of 0.83, as shown in Figure 4.3. Occasionally some benchmarks increased the number of return address assignments, which can occur if we hoist a return address assignment for a loop that does not execute. The graph shows that about 17% of the calls are in loops with one call or multiple calls with a common return address when a register is available to hoist the return address assignment.

The SP ToC optimizations improved the execution cycle ratio, depicted in Figure 4.4, from an average of 0.99 to 0.94. Most of the benchmarks improved, though the performance *fft* and *ispell* was slightly degraded.

Figure 4.5 shows that our ToC optimizations resulted in a small increase in code size from 0.913 without ToC optimizations to 0.922 with ToC optimizations. Note some ToC optimizations decreased code size while others increased it.

Figure 4.6 shows the results of our simulations on estimated energy usage. On average, the SP reduces energy usage by 20.0%. These savings comes primarily from the reduction in register file accesses, branch prediction table accesses, and the fact that we do not need a branch target buffer.

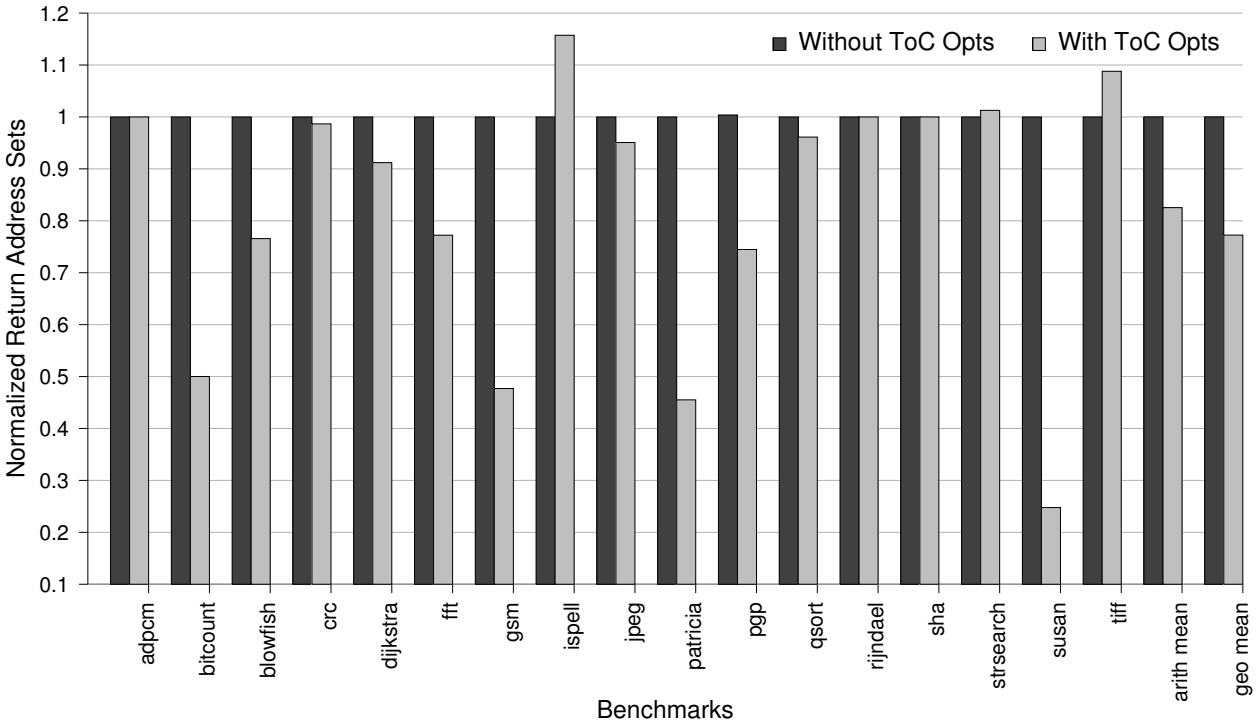

Figure 4.3: Return Address Assignment Ratio

Of course these results are also affected by the relative running time of the benchmark as that has a direct effect on instruction cache usage and static power consumption. While these estimates take into account the number of accesses to the larger structures of the two pipelines the difference in control logic and interconnect routing is not taken into account. Applying ToC optimizations decreases energy usage by an additional 3.4%.

Figures 4.7 and 4.8 show the impact of each ToC optimization on execution time and energy usage, respectively, where a ToC optimization is added to the previous set applied. The execution cycle ratio is largely affected by using the `SEQ` register to hoist the target address calculation of the top-most block in an innermost loop. There are many applications where most of the execution cycles are spent in innermost loops that do not have any other conditional control flow. Hoisting other target address calculations out of loops provided an additional 0.4% reduction. The execution time benefits for this optimization were also limited by only eliminating one effect for each ToC rather than eliminating two effects for each ToC when using the SEQ register. Call-jump/jump-call chaining provided only a small benefit, which was primarily due to the infrequency of unconditional

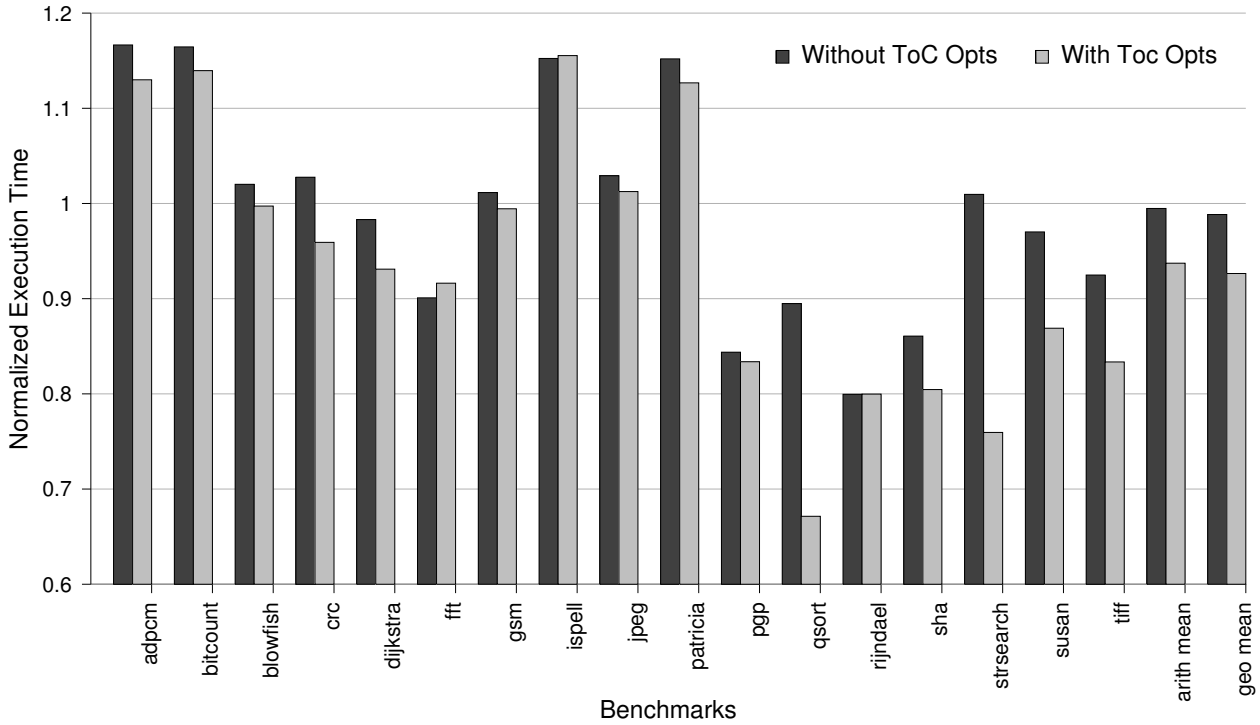

Figure 4.4: Execution Cycle Ratio

jumps that limited the opportunities for this optimization to be applied. Hoisting return address assignments provided a 0.2% benefit. Most of the 0.4% benefit for exploiting conditional calls/returns was due to merging calls before branches, as depicted in Figure 3.9. Call-jump/jump-call chaining, hoisting return address assignments, and exploiting conditional calls all require the invocation of a function and thus their benefits are limited due to the relative execution time of the invoked function. Conditional returns were also infrequently applied. The impact of ToC optimizations on energy usage is highly correlated to the improvements for execution time. Note that much of the 20% reduction with no ToC optimizations is achieved by the way that ToCs are performed, which eliminates the need for a BTB and RAS and significantly decreases BPB accesses.

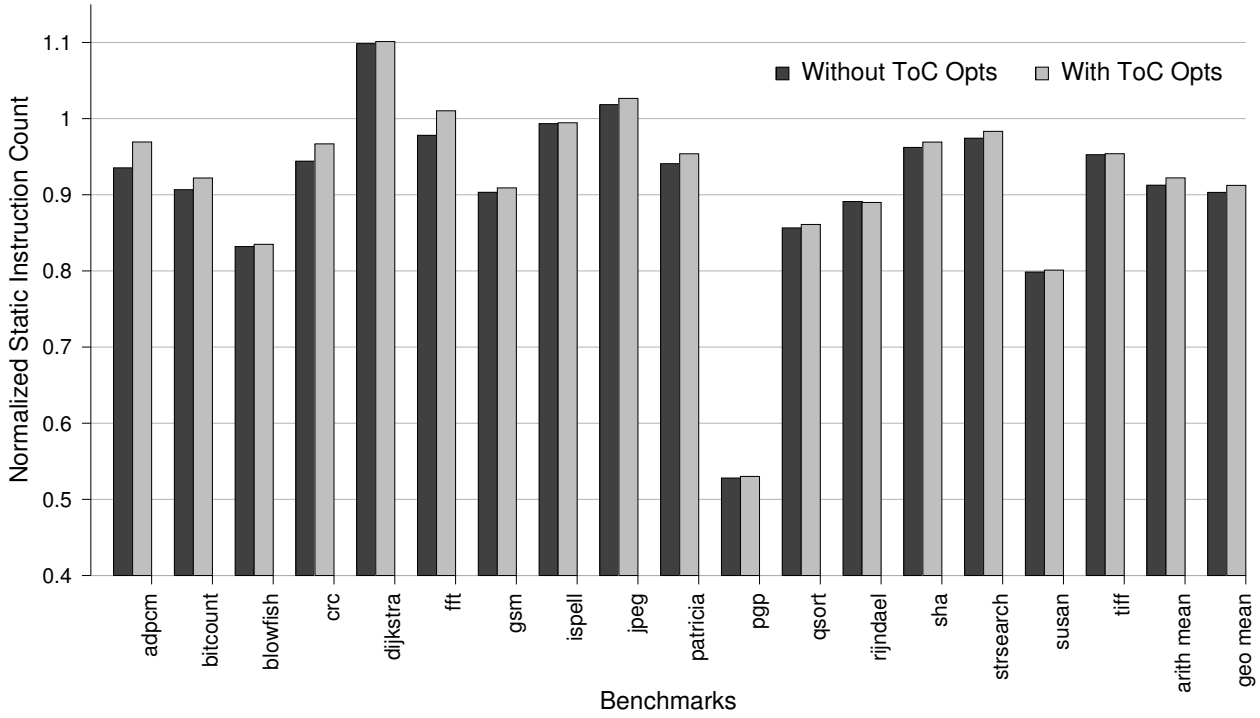

Figure 4.5: Code Size Ratio

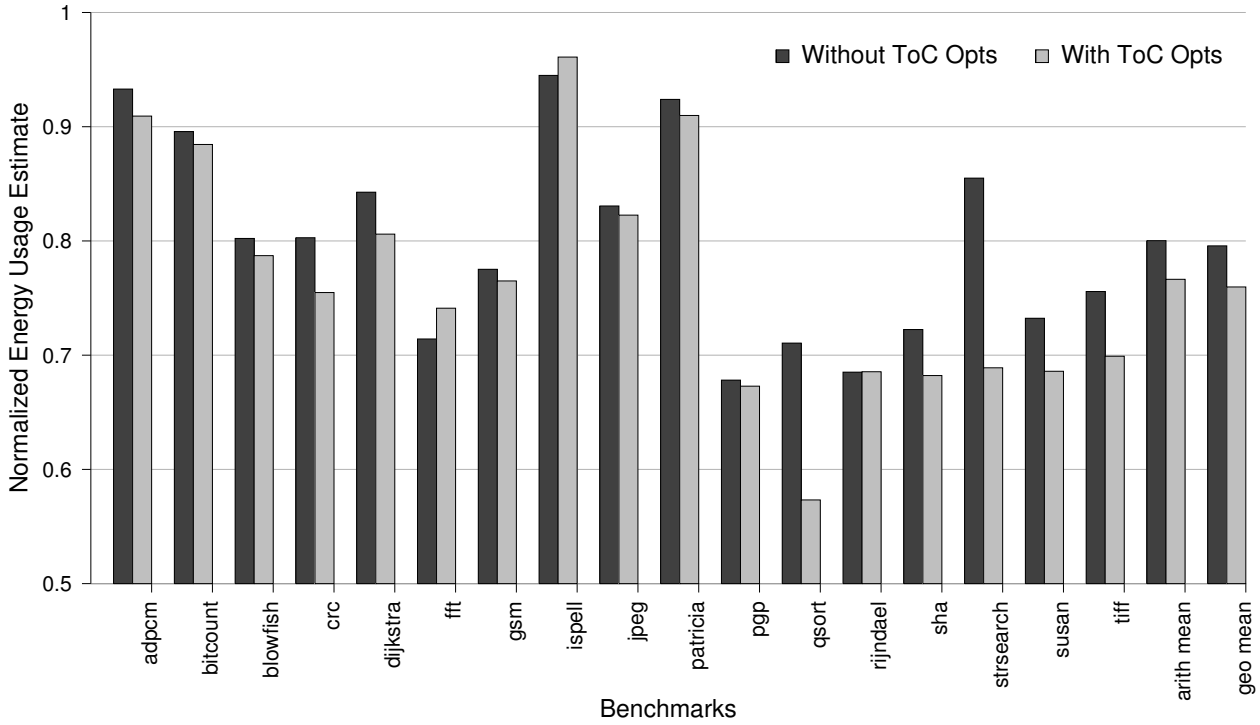

Figure 4.6: Estimated Energy Usage Ratio

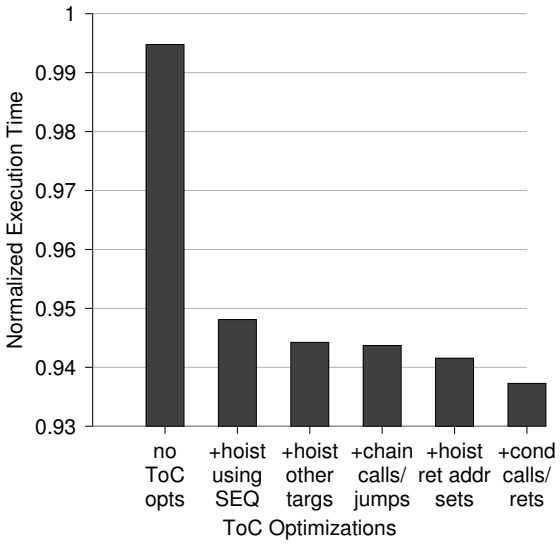

Figure 4.7: Impact of ToC Optimizations on Execution Time

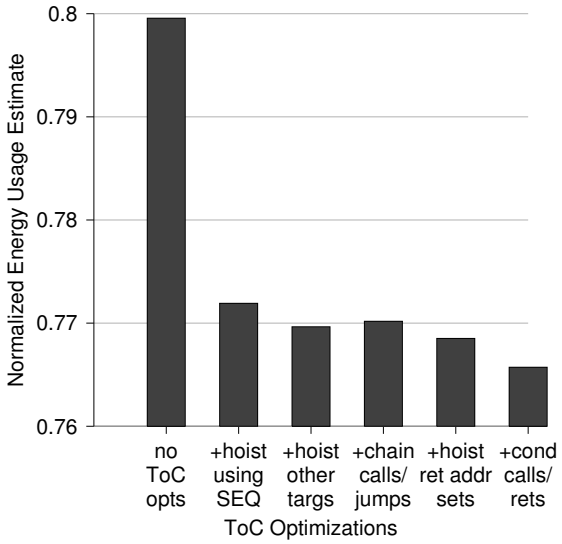

Figure 4.8: Impact of ToC Optimizations on Energy Usage

CHAPTER 5

CONCLUSIONS

Processors perform a significant number of ToCs and often use auxiliary hardware structures (BTB, RAS, and BPB) to quickly perform ToCs. In micro-effect based architectures, it makes sense to reconsider the way branches are handled. ToC operations on the SP architecture are separated into multiple effects that eliminate the need for a BTB or RAS, significantly decrease the number of BPB accesses, and provide opportunities for the compiler to perform additional ToC optimizations. Many of the target address calculations performed are redundant as direct targets do not change when they are repeatedly calculated. For the SP architecture, these target address calculations can be hoisted out of loops or eliminated when the target address is already available. Likewise, branch chaining can be performed between calls and jumps, return address assignments can be hoisted out of loops, and conditional calls and returns can be exploited. We have shown in this paper that the low-level SP representation enables a compiler to more effectively optimize ToCs and provides improvements in both performance and energy usage.

REFERENCES

[1] R. Baird, P. Gavin, M. Själander, D. Whalley, and G. Uh. Optimizing Transfers of Control in the Static Pipeline Architecture. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 1–10, 2015.

[2] B. Davis, P. Gavin, R. Baird, M. Sjalander, I. Finlayson, F. Rasapour, G. Cook, G. Uh, D. Whalley, and G. Tyson. Scheduling Instruction Effects for a Statically Pipelined Processor. In *Proceedings of the IEEE/ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 167–176, 2015.

[3] C.W. Fraser. A retargetable compiler for ansi c. *ACM Sigplan Notices*, 26(10):29–43, 1991.

[4] M.E. Benitez and J.W. Davidson. A Portable Global Optimizer and Linker. *ACM SIGPLAN Notices*, 23(7):329–338, 1988.

[5] I. Finlayson, G. Uh, D. Whalley, and G. Tyson. An Overview of Static Pipelining. *Computer Architecture Letters*, 11(1):17–20, 2012.

[6] I. Finlayson, B. Davis, P. Gavin, G. Uh, D. Whalley, M. Själander, and G. Tyson. Improving processor efficiency by statically pipelining instructions. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA, June 2013. ACM.

[7] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2002.

[8] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[9] S. Thoziyoor, N. Muralimanohar, J.H Ahn, and N.P Jouppi. Cacti 5.1. Technical report, HP Laboratories, Palo Alto, April 2008.

BIOGRAPHICAL SKETCH

The author was born in Boise, Idaho. The author attended high school at Meridian Technological Charter High school where he began a hobby of programming. While attending high school, the author started working in web development. After graduating high school in 2009, the the author persued post-secondary education at Boise State University. While attending school at BSU, the author picked up another job as an undergraduate research assistant, and presented at the LLVM compiler conference in 2013. The author received a Bachelor of Science in Computer Science at BSU with the Outstanding Graduating Senior Award. The author subsequently stopped working as a web developer and undergraduate research assistant, and began attending graduate school at Florida State University with the plan of first pursuing a Masters Degree and then continuing to pursue Doctoral degree. During the time spent pursuing a Master's Degree, the author has two publications, and was the primary author on one of those publications, "Optimizing Transfers of Control in the Static Pipeline Architecture" [1].