

Optimizing Transfers of Control in the Static Pipeline Architecture

Ryan Baird
Peter Gavin

Florida State University
Tallahassee, Florida, USA
baird@cs.fsu.edu, pgavin@gmail.com

Magnus Sjalander

Uppsala University
Uppsala, Sweden
magnus.sjalander@it.uu.se

David Whalley
Gang-Ryung Uh *

Florida State University
Tallahassee, Florida, USA
whalley@cs.fsu.edu, uryung@gmail.com

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 more 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.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors- compilers, optimization

General Terms Algorithms, Measurements, Performance.

Keywords Transfers of Control, Compiler Optimizations, Energy Efficiency.

1. Introduction

Power and energy are now critical design constraints in processors for several reasons. 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 a more power efficient processor helps to address all of these issues.

The static pipeline (SP) is a recent approach to processor design that 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 the first detailed description of how ToCs are implemented in the SP architecture, (3) implementing a variety of new SP ToC optimizations that exploit the decoupling of ToC effects in a fully exposed datapath that would not otherwise be possible, and (4) evaluating the impact of these SP ToC optimizations.

2. Overview of Static Pipeline Architecture

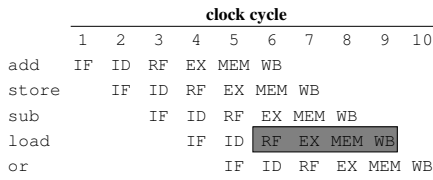
Figure 1 illustrates the basic idea of the SP approach as compared to a traditional architecture. Each instruction spends several cycles within the processor with traditional pipelining. For example, the *load* instruction in Figure 1(b) requires one cycle for each stage after being fetched and decoded and remains in the pipeline from cycles six through nine. Figure 1(c) illustrates how an SP processor operates. Conventional operations, such as a load, still require multiple cycles to complete. However, these conventional operations are spread over multiple SP instructions, which are performed in three stages: fetch, decode, and execute. Each SP instruction specifies how all the remaining portions of the processor besides fetch and decode are controlled during the cycle it is executed. An SP instruction consists of one or more effects. In the execute stage, the SP processor executes each of these instruction effects in parallel. Essentially, an SP instruction controls a set of parallel effects during a single cycle instead of a sequence of interdependent effects.

A high-level overview of the datapath for our current SP design is shown in Figure 2. There are ten internal registers that are explicitly accessed within SP instructions. The SEQ (sequential address) register is assigned 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

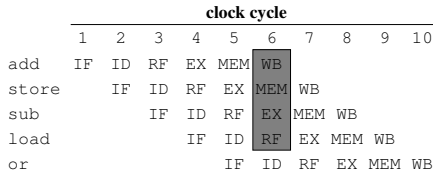
* Currently with Qualcomm Technologies, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCTES'15, June 18–19, 2015, Portland, Oregon, USA.
Copyright © 2015 ACM 978-1-4503-3257-6/...\$15.00
<http://dx.doi.org/10.1145/2670529.2754952>



(a) Traditional Pipelining



(b) Static Pipelining

Figure 1: Traditionally Pipelined vs. Statically Pipelined Instructions

these internal registers are small and can be placed near the portion of the processor that accesses them. Since these internal registers are explicitly accessed in SP instructions, a new level of compiler optimizations can be exploited.

To better illustrate the mechanics of an SP instruction, it is helpful to look at an example:

```
RS1=r[3]; OPER1=RS1+SE; SE=4;
```

This instruction, in parallel, (1) reads the integer register $r[3]$, (2) adds the values of internal registers $RS1$ and SE , and (3) sign extends the value 4. At the end of the cycle, it stores the result of (1) into $RS1$, (2) into $OPER1$, and (3) into SE . Forwarding logic is eliminated as the compiler explicitly controls which internal register values are to be read and hazard detection is simplified compared to a conventional pipeline.

Unfortunately, allowing for every possible combination of effects in an SP instruction would require more than 80 bits. To keep the instruction size small, we use a template based instruction encoding with the formats shown in Figure 3 by selecting the most frequent combinations of effects to be encoded into instructions. Essentially, we compile benchmarks allowing all instructions that could possibly fit in one of the templates and then select the 32 most frequently used instructions.

5-bit ID	Long Immediate			10-bit Field			
5-bit ID	Long Immediate			3-bit	7-bit Field		
5-bit ID	Long Immediate			2-bit	4-bit Field	4-bit Field	
5-bit ID	10-bit Field		7-bit Field		10-bit Field		
5-bit ID	10-bit Field		7-bit Field		3-bit	7-bit Field	
5-bit ID	10-bit Field		7-bit Field		2-bit	4-bit Field	4-bit Field
5-bit ID	7-bit Field	7-bit Field	7-bit Field	7-bit Field	4-bit Field	2-bit	

10-bit Effect	7-bit Effect	4-bit Effect	3-bit Effect	2-bit Effect
ALU Operation	Integer Addition	Move to CP1/2	SEQ or SE	SEQ Set
FPU Operation	Load Operation	Prepare to Branch		Move CP1/2 to SE
Load or Store Operation	Single Register Read			
Dual Register Reads	Short Immediate			
Register Write				

Figure 3: Static Pipeline Template Formats

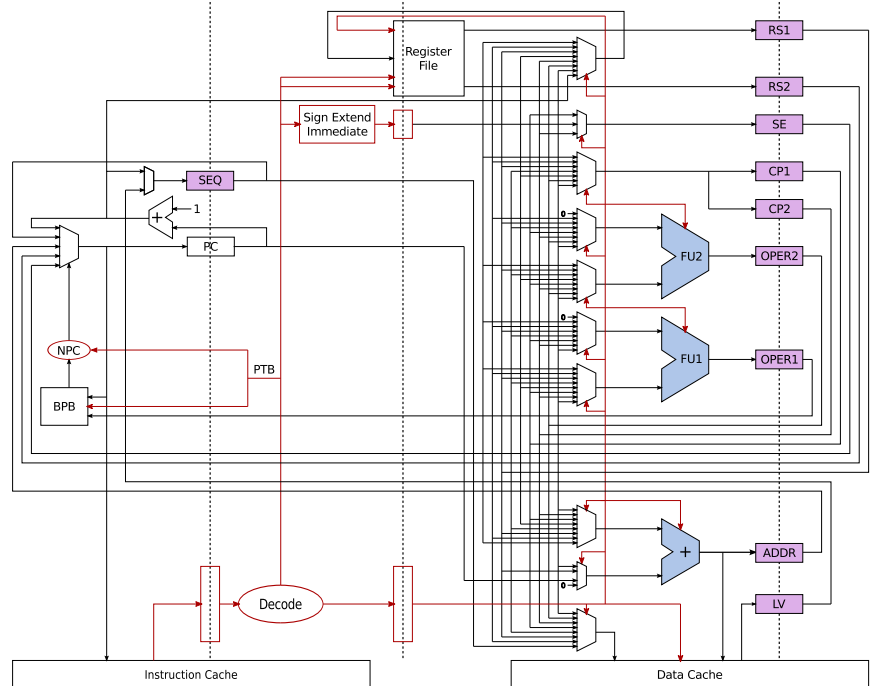


Figure 2: Static Pipeline Datapath

3. Changes to the Static Pipeline Architecture

The SP pipeline in our previous simulations consisted of two stages, instruction fetch (IF) and execute (EX), where decoding instructions was assumed to be part of the EX stage [7]. In order to make the clock period comparable to a baseline MIPS processor in a new SP VHDL implementation, 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 both data memory address calculations and 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), which sometimes requires an extra effect to move an address to the ADDR register. The result 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.

4. Static Pipeline Transfers of Control

ToC operations (branches, jumps, calls, returns) in an SP architecture are explicitly separated into three parts that spans multiple SP instructions: (1) the target address calculation, (2) the prepare to branch (PTB) command, and (3) the point of the ToC. Figure 4 provides examples of how ToCs are accomplished in the SP architecture. Most SP target address calculations are initially accomplished by either calculating the sum of 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 4(a) and 4(b), respectively. At some point 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 also 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);      SE=LO(label);
ADDR=PC+SE;            SE=SE|HI(label);
...                    ...
PTB=c:ADDR;           PTB=u:SE;
PC=OPER1!=RS1,ADDR;   PC=SE;r[31]=PC+1;
(a) PC-Relative SP Branch  (b) Absolute SP Call

```

Figure 4: SP Transfer of Control Examples

Figure 5 contains a pipeline diagram showing how the SP instructions involving the 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 5. 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 5 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 5.

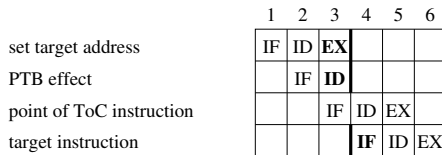


Figure 5: Pipelining SP Transfers of Control

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 and contain target addresses and tags to confirm the instruction being fetched is the correct ToC. The target addresses in the BTB and RAS are not needed in an SP processor since the target address is always known before the target instruction is fetched. The BTB and RAS tags are not needed since the PTB effect identifies that the following instruction is a ToC. Thus, the BTB and RAS are completely removed in the SP architecture. 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 BPB in a conventional processor is also always accessed in the IF stage and contains bits to indicate if the branch is predicted to be taken or not taken. Conditional ToCs in an SP processor are indicated in the PTB effect that immediately precedes the point of the ToC, so the BPB is only accessed for conditional SP ToCs.

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.

5. 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 6 shows the steps of our compilation process. First, C code is input to the *frontend*, which consists of the LCC compiler [10] frontend that converts LCC's output format into the register transfer list (RTL) format used by the VPO compiler [2].

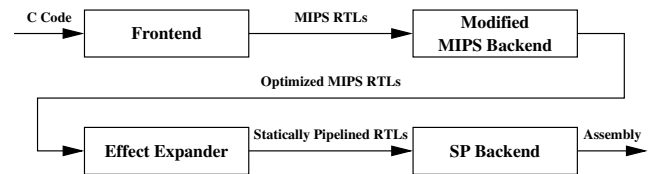


Figure 6: SP Compilation Process

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 duplicating 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.

6. 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. This is the first paper that describes how any SP ToC optimizations are accomplished. With the exception of the SEQ hoisting optimization, all of these optimizations are new as compared to prior SP studies [6–8].

6.1 Hoisting the Top of Innermost Loop Target

Since address calculations are typically tightly coupled with ToC instructions in a conventional ISA, they are recalculated every loop iteration, even though the target address does not change. For PC-relative ToCs, this means an 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. 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 instructions that perform ToCs in an application are typically in the innermost loops of functions. The SP architecture provides the SEQ internal register to hold the address of the top-most instruction of an innermost loop. The only 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 example in Figure 7 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 shown in this paper. 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 7) being assigned to the SEQ register. Note the top-most block in the loop is not always the loop header, but it 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 can potentially improve performance due to possibly decreasing the number of instructions

in the loop after scheduling SP effects. For instance, Figure 7(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 7(b).

<pre> 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); </pre>	<pre> SEQ=PC+1; L1 # Beginning of loop ... PTB=c:SEQ; PC=RS2==LV, SEQ(L1); ... PTB=c:SEQ; PC=OPER1!=CP1, SEQ(L1); </pre>
(a) Before SEQ Optimization	(b) After SEQ Optimization

Figure 7: Example of SEQ Optimization

6.2 Hoisting Other Target Address Calculations

There are often 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 [7], 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 when an integer register 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 8, 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 3). 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 3). 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 ToC since RS2 is used to read the register value and serve as the target specified in the 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 8: Algorithm for Hoisting Target Address Calculations

Figure 9 depicts an example of applying this optimization within a loop nest. Figure 9(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 9(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 9(c) shows the SP instructions after applying this optimization. The target address calculation of `L3` 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 originally only used by the compiler to store the return address into `r[31]` for a call. Now we assign `PC+1` to a register to obtain 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 `L3`, where two distinct calculations are required in the loop in Figure 9(b) if the values of `SE` or `ADDR` are overwritten between the two ToCs. The second read of `r[17]` in Figure 9(c) will be eliminated after common subexpression elimination is applied if `RS2` is not overwritten after the first read of `r[17]`. 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 possibly performance.

6.3 Performing Call-Jump and Jump-Call Chaining

Sometimes a call is followed by an unconditional jump or the target block of an unconditional jump contains a call. In these situations the unconditional jump can be eliminated by adjusting the return address register assignment associated with the call.

Figure 10 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 10(a) shows a call to `f1` in the `then` portion of an `if-then-else` statement. As shown in Figure 10(b), the call to `f1` is followed by an unconditional jump to `L2` that jumps over the `else` portion of the statement. Figure 10(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.

```

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

```

(a) Loop at Source Code Level

<pre> 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); </pre>	<pre> 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); </pre>
(b) Loop after SEQ Transformation	(c) Loop after Hoisting other Target Address Calculations

Figure 9: Example of Target Address Calculation Hoisting

```

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

```

(a) Call Followed by Jump at Source Code Level

<pre> SE=LO:f1; SE=SE HI:f1; # call to PTB=u:SE; # f1 PC=SE(f1);r[31]=PC+1; ... SE=offset(L2); ADDR=PC+SE; # jump PTB=u:ADDR; # over PC=ADDR(L2); # else </pre>	<pre> ... SE=offset(L2); ADDR=PC+SE; r[31]=ADDR; # call to SE=LO:f1; # f1 SE=SE HI:f1; # with PTB=u:SE; # return PC=SE(f1); # to L2 </pre>
(b) Before Chaining Call to Jump	(c) After Chaining Call to Jump

Figure 10: Example of Call-Jump Chaining

Figure 11 shows how a jump is eliminated when a jump is followed by a call. Figure 11(a) shows a call to `f2` after an `if-then-else` statement. As shown in Figure 11(b), the target block, `L4`, of the unconditional jump contains a call to `f2`. Figure 11(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

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

(b) Before Chaining Jump to Call (c) After Chaining Jump to Call

Figure 11: 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.

6.4 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 12 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.

```

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 12: Hoisting Return Address Assignments Algorithm

Figure 13 depicts an example of this transformation performed on multiple calls within a single loop. Figure 13(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 13(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 13(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 address hoisting transformation reduces the overall code size, the number of ToCs executed, and the energy usage required to execute the code.

6.5 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.

Figure 14 depicts an example of exploiting a conditional return. Figure 14(a) shows a source code fragment and Figure 14(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 14(c) shows the SP instructions after performing the optimization. The branch target is set to the return address and the original target address calculation of L4 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.

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 pre-

```

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; # f1
PC=RS2(f1); r[31]=PC+1;
RS2=r[20];
PTB=u:RS2; # jump to
PC=RS2(L3); # L3
L2 ...
RS2=r[19]; # call to
PTB=u:RS2; # 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 13: Example of Return Address Assignment Hoisting

```

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

```

(a) Branch Followed by a Return at the 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 a 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 a Conditional Return

Figure 14: Example of Exploiting a Conditional Return

cedes a branch and the effects between the call and the branch can be moved before the call. Figure 15 depicts an example of exploiting a conditional call. Figure 15(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 15(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 15(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

Table 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

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
...
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 15: Example of Exploiting a Conditional Call

7. Evaluation

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

7.1 Experimental Setup

We use 17 benchmarks shown in Table 1 from the MiBench benchmark suite [11], which is a representative set of embedded applications. We use an extended GNU assembler to assemble SP instructions and a simulator based on the SimpleScalar in-order MIPS [1]. For all benchmarks, when compiled for the SP, over 90% of the instructions executed are SP instructions, with the remaining MIPS instructions coming from calls to standard library routines such as *printf*. All cycles and register accesses are counted towards the results whether they come from the MIPS library code or the SP code.

For the MIPS baseline, the programs were compiled with the original VPO MIPS port with all optimizations enabled and run through the same simulator, as it is also capable of simulating MIPS code. We extended the simulator to include branch prediction with a simple bimodal branch predictor with 256 two-bit saturating counters, and a 256-entry branch target buffer. The branch target buffer (BTB) is only used for MIPS code as it is not needed for the SP. The simulator was also extended to include level one data and instruction caches, which were configured to have 256 lines of 32 bytes each and are direct-mapped.

7.2 Results

Each of the graphs in this section 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.

There are more PC-relative calculations in the initial SP code since these calculations are sometimes speculatively performed due to cross block scheduling and unconditional jumps on the MIPS were performed using absolute addresses and SP unconditional jumps used PC-relative target addresses. PC-relative target address calculations improved from a ratio of 1.04 to a ratio of 0.44, as shown in Figure 16. This improvement was primarily due to both utilizing the *SEQ* register and the integer register file to hoist target address calculations out of loops. Note the MIPS ISA does not provide a way to perform a conditional branch with the target in a register.

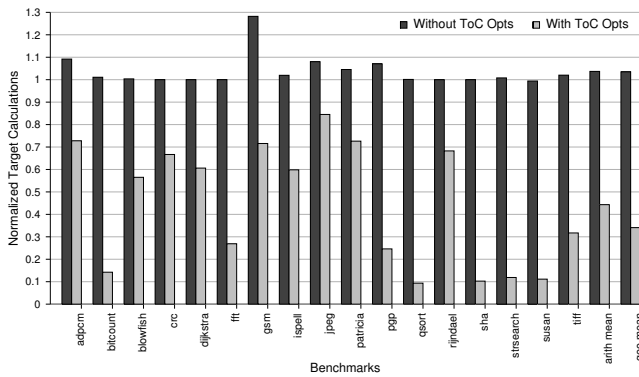


Figure 16: PC-Relative Target Address Calculation Ratio

The number of absolute target address calculations improved from a ratio of 1.00 to a ratio of 0.87, as shown in Figure 17. This improvement came from performing fewer target address calculations of calls inside loops. This improvement is less than the improvement for PC-relative addresses since direct calls are not always in loops while at least one conditional branch will be in each loop and the *SEQ* hoisting optimization does not hoist absolute (call) target address calculations. Likewise, the presence of a call in a loop requires 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.

On average, we were able to reduce the number of return address assignments to a ratio of 0.83, as shown in Figure 18. Occasionally some benchmarks increased the number of return address assignments, which occurs when a loop immediately exits after being entered. This occurred for *ispell*, which performs a number of string comparisons where the first character in each string may differ. The results indicate 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 19, from an average of 0.99 to 0.93. All of the benchmarks improved in performance relative to not performing ToC optimizations.

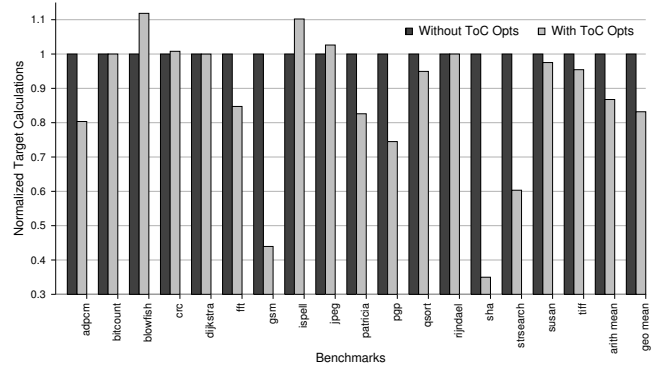


Figure 17: Absolute Target Address Calculation Ratio

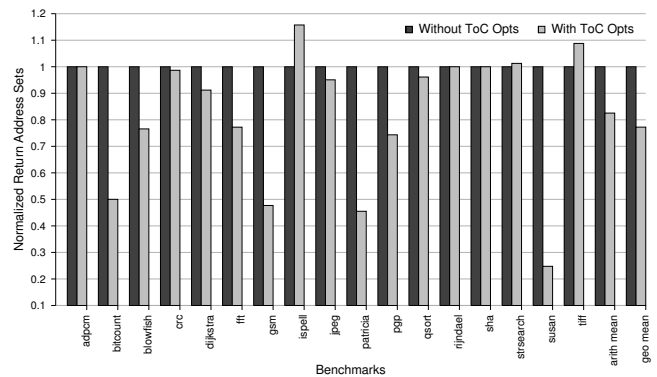


Figure 18: Return Address Assignment Ratio

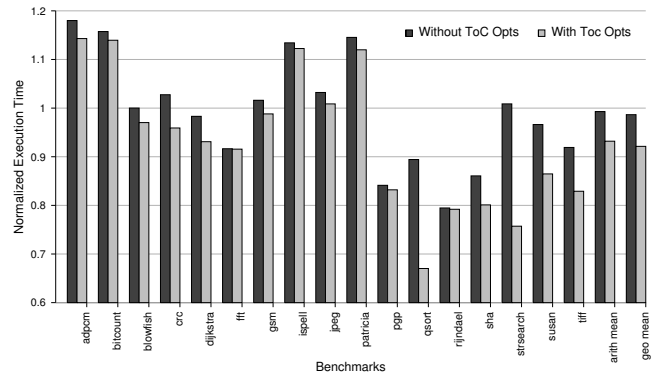


Figure 19: Execution Cycle Ratio

Figure 20 shows that our ToC optimizations resulted in a small increase in code size from 0.914 without ToC optimizations to 0.922 with ToC optimizations. Note some ToC optimizations can decrease the code size while others, such as hoisting target address calculations, typically increased it.

We also present an estimate of the processor energy savings achieved by the SP approach. We use the simulated counts of events such as register file accesses, branch predictions and ALU operations along with estimates of how much power is consumed by each event. The SRAMs within the pipeline have been modeled using CACTI [13]. Other components have been synthesized for a 65nm process, then simulated at the netlist level to determine average case activation power. We have normalized the power per component to a 32-entry dual-ported register file read, because the

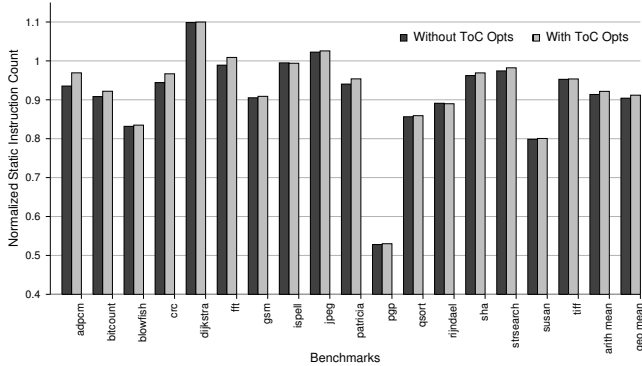


Figure 20: Code Size Ratio

power per component are dependent on process technology and other implementation dependent issues. The ratios between component power are also somewhat dependent on process technology, however these differences should not have a qualitative impact on the final estimates. The resulting total energy estimate is a linear combination of the number of activations and the power attributions per component. The relative power per activation we attribute to each component is given in Table 2.

Table 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

Figure 21 shows the results of our simulations. On average, the SP reduces energy usage by 20.2%. 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. 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 3.6%.

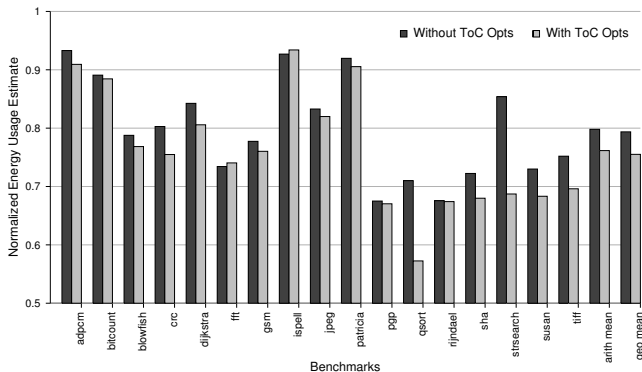


Figure 21: Estimated Energy Usage Ratio

Figures 22 and 23 show the impact of the ToC optimizations on execution time and energy usage, respectively, where a ToC optimization is added to the previous set applied. The execution time 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 0.8% 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 jumps that limited the opportunities for this optimization to be applied. Hoisting return address assignments also provided a small benefit. Most of the benefit for exploiting conditional calls/returns was due to merging calls before branches, as depicted in Figure 15. 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 all of the call and return related optimizations together was only about 0.24%. We anticipate that these call and return related ToC optimizations will be more effective for applications that are more call and return intensive, such as object-oriented applications in C++. The impact of ToC optimizations on energy usage is highly correlated to the improvements for execution time. Note that much of the 20% energy usage 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.

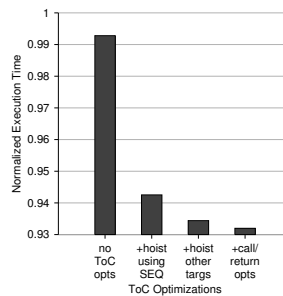


Figure 22: Impact of ToC Opts on Execution Time

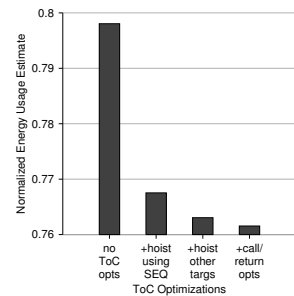


Figure 23: Impact of ToC Opts on Energy Usage

8. Related Work

SP instructions are most similar to horizontal microinstructions [15], however, there are significant differences. While horizontal microinstructions also specify computation at a low level, they do not expose pipelining at the architectural level. In a micro-programmed processor, each machine instruction causes the execution of microinstructions within a micro-routine stored in ROM. Furthermore, compiler optimizations cannot be performed across these micro-routines since this level is not generally exposed to the compiler. Static pipelining also bears some resemblance to VLIW [9] in that the compiler determines which operations are independent. However, most VLIW instructions represent multiple RISC operations that can be performed in parallel. In contrast, the SP approach encodes individual instruction effects that can be issued in parallel, where most of these effects correspond to an action taken by a single pipeline stage of a traditional RISC instruction.

A prepare-to-branch (PTB) instruction has been previously proposed [3]. However, the use of this feature required an entire instruction and thus will impact code size and performance. In contrast, our PTB field only requires 4 bits as the target address calculation is decoupled from both the PTB field and the point of the ToC.

One study proposed to use a set of branch registers to hold branch target addresses, a set of instruction registers to hold branch target instructions, and to have every instruction reference a branch register (be a transfer of control), where one branch register represented the next sequential address [5]. The proposed approach enables target address calculations to be hoisted out of loops or eliminated by common subexpression elimination. However, their technique still requires the use of a delayed branch and that each assignment to a branch register cause a prefetch of the branch target instruction into an instruction register in order to make each taken ToC feasible without a delay. This prefetching requires a multi-ported instruction cache, which would increase the L1 IC complexity, area, and access power. This study also did not utilize the other optimizations that we present in our paper.

There have been other proposed architectures that also expose much of the datapath to a compiler. The No Instruction Set Computer (NISC) [12] has no fixed ISA that bridges the compiler with the hardware. Instead, the compiler generates control signals for the datapath directly. The FlexCore processor [14] also exposes datapath elements at the architectural level. The design features a flexible datapath with an instruction decoder that is reconfigured dynamically at runtime. The Transport-Triggered Architectures (TTAs) [4] are similar to VLIWs in that there are a large number of parallel computations specified in each instruction. TTAs, however, can move values directly to and from functional unit ports, to avoid the need for large, multi-ported register files. However, the SP backend performs many other optimizations that are not performed for the the NISC, FlexCore, and TTA architectures while using fewer internal registers. The NISC, FlexCore, and the initial TTA studies improve performance at the expense of a significant increase in code size and were evaluated using tiny benchmarks. In contrast, static pipelining focuses on improving energy usage while still obtaining performance and code size improvements on the MiBench benchmark suite. In addition, the NISC, FlexCore, and TTA rely on delayed branches, where the SP decouples the branch target address calculation from the branch and uses a PTB field, completely eliminating the need for a BTB, which is the most expensive part of branch prediction. Finally, many new ToC optimizations are performed for the SP that are not performed for these other fully-exposed datapath architectures.

9. 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 by traditional architectures 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.

10. Acknowledgements

We appreciate the comments provided by the anonymous reviewers of this paper. This research was supported in part by the US National Science Foundation grants CNS-0964413 and CNS-0915926 and the Korean Ministry of Science, ICT and Future Planning grant 10041725.

References

- [1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.
- [2] M. Benitez and J. Davidson. A Portable Global Optimizer and Linker. *ACM SIGPLAN Notices*, 23(7):329–338, 1988.
- [3] A. Bright, J. Fritts, and M. Gschwind. Decoupled fetch-execute engine with static branch prediction support. Technical report, IBM Research Report RC23261, IBM Research Division, 1999.
- [4] H. Corporaal and M. Arnold. Using Transport Triggered Architectures for Embedded Processor Design. *Integrated Computer-Aided Engineering*, 5(1):19–38, 1998.
- [5] J. Davidson and D. Whalley. Reducing the cost of branches by using registers. In *International Symposium on Computer Architecture*, pages 182–191, May 1990.
- [6] I. Finlayson, G. Uh, D. Whalley, and G. Tyson. An Overview of Static Pipelining. *Computer Architecture Letters*, 11(1):17–20, 2012.
- [7] Finlayson, I. and Davis, B. and Gavin, P. and Uh, G. and Whalley, D. and Sjalander, M. and Tyson, G. Improving Processor Efficiency by Static Pipelining Instructions. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 33–43, 2013.
- [8] Finlayson, I. and Uh, G. and Whalley, D. and Tyson, G. Improving Low Power Processor Efficiency with Static Pipelining. In *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures*, 2011.
- [9] J. Fisher. VLIW Machine: A Multiprocessor for Compiling Scientific Code. *Computer*, 17(7):45–53, 1984.
- [10] C. Fraser. A retargetable compiler for ansi c. *ACM Sigplan Notices*, 26(10):29–43, 1991.
- [11] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. 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.
- [12] M. Reshadi, B. Gorjara, and D. Gajski. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *International Conference on Computer Design*, pages 69–76, Washington, DC, USA, 2005.
- [13] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi. Cacti 5.1. Technical report, HP Laboratories, Palo Alto, Apr. 2008.
- [14] M. Thuresson, M. Sjalander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. Flexcore: Utilizing exposed datapath control for efficient computing. *Journal of Signal Processing Systems*, 57(1):5–19, 2009.
- [15] M. Wilkes and J. Stringer. Micro-Programming and the Design of the Control Circuits in an Electronic Digital Computer. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 49, pages 230–238. Cambridge Univ Press, 1953.