Improving Processor Efficiency by Statically Pipelining Instructions

Ian Finlayson
University of Mary Washington
Fredericksburg
Virginia, USA
finlayson@umw.edu

Brandon Davis Peter Gavin
Florida State University
Tallahassee
Florida, USA
{bdavis,gavin}@cs.fsu.edu

Gang-Ryung Uh
Boise State University
Boise
Idaho, USA
uh@cs.boisestate.edu

David Whalley Magnus Själander Gary Tyson
Florida State University
Tallahassee
Florida, USA
{whalley,sjalande,tyson}@cs.fsu.edu

Abstract

A new generation of applications requires reduced power consumption without sacrificing performance. Instruction pipelining is commonly used to meet application performance requirements, but some implementation aspects of pipelining are inefficient with respect to energy usage. We propose static pipelining as a new instruction set architecture to enable more efficient instruction flow through the pipeline, which is accomplished by exposing the pipeline structure to the compiler. While this approach simplifies hardware pipeline requirements, significant modifications to the compiler are required. This paper describes the code generation and compiler optimizations we implemented to exploit the features of this architecture. We show that we can achieve performance and code size improvements despite a very low-level instruction representation. We also demonstrate that static pipelining of instructions reduces energy usage by simplifying hardware, avoiding many unnecessary operations, and allowing the compiler to perform optimizations that are not possible on traditional architectures.

1. Introduction

Energy expenditure is clearly a primary design constraint, especially for embedded processors where battery life is directly related to the usefulness of the product. As these devices become more sophisticated, the execution performance requirements increase. This trend has led to new generations of efficient processors that seek the best solution to the often conflicting requirements of low-energy design and high-performance execution. Many of the micro-architectural techniques to improve performance were developed when efficiency was not as important. For instance, speculation is a direct tradeoff between power and performance, but many other techniques are assumed to be efficient.

Instruction pipelining is one of the most common techniques for improving performance of general-purpose processors. Pipelining is generally considered very efficient when speculation costs and scheduling complexity are minimized. While it is true that speculation, dynamic scheduling policies, and superscalar execution have the largest impact on efficiency, even simple, in-order, scalar pipelined architectures have inefficiencies that lead to less optimal implementations of the processor architecture. For instance, hazard detection and data forwarding not only require evaluation of register dependencies each cycle of execution, but successful forwarding does not prevent register file accesses to stale values, nor does it eliminate unnecessary pipeline register writes of those stale values, which are propagated for all instructions.

The goal of this paper is to restructure the organization of a pipelined processor implementation in order to remove as many redundant or unnecessary operations as possible. This goal is achieved by making the pipelined structures architecturally visible and relying on the compiler to optimize resource usage. While techniques like VLIW [8] have concentrated on compile time instruction scheduling and hazard avoidance, we seek to bring pipeline control further into the realm of compiler optimization. When pipeline registers become architecturally visible, the compiler can directly manage tasks like forwarding, branch prediction, and register access. This mitigates some of the inefficiencies found in more conventional designs, and provides new optimization opportunities to improve the efficiency of the pipeline.

Figure 1 illustrates the basic idea of the static pipeline (SP) approach. With traditional pipelining, each instruction spends several cycles in the pipeline. For example, the `load` instruction in Figure 1(b) requires one cycle for each stage and remains in the pipeline from cycles five through eight. Each instruction is fetched and decoded and information about the instruction flows through the pipeline, via pipeline registers, to control each portion of the processor that takes a specific action during each cycle. The `load` instruction shares the pipelined datapath with other instructions that are placed into the pipeline in adjacent cycles.

LCTES'13, June 20–21, 2013, Seattle, Washington, USA.

	Traditional: clock cycle									Static: clock cycle								
	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
`add`	IF	RF	EX	MEM	WB					IF	RF	EX	MEM	WB				
`store`		IF	RF	EX	MEM	WB					IF	RF	EX	MEM	WB			
`sub`			IF	RF	EX	MEM	WB					IF	RF	EX	MEM	WB		
`load`				IF	RF	EX	MEM	WB					IF	RF	EX	MEM	WB	
`or`					IF	RF	EX	MEM	WB					IF	RF	EX	MEM	WB
(a) Traditional Insts	**(b) Traditional Pipelining**									**(c) Static Pipelining**								

Figure 1. Traditionally Pipelined vs. Statically Pipelined Instructions

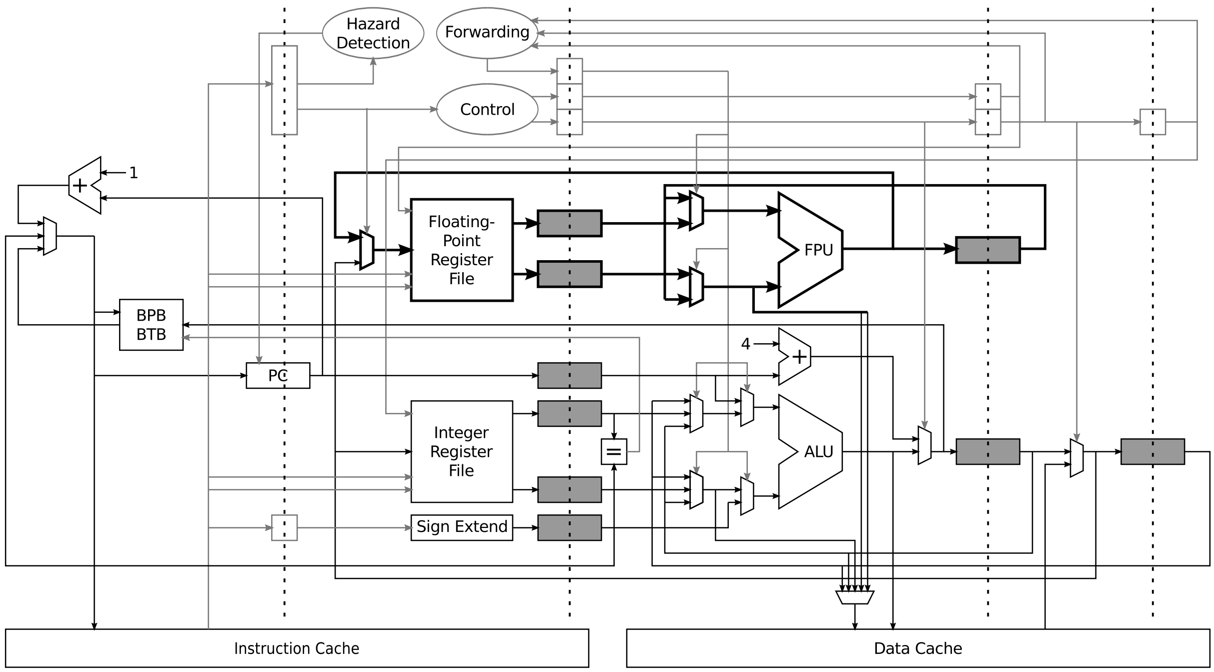

Figure 2. Classical Five-Stage Pipeline

Figure 1(c) illustrates how an SP processor operates. Operations associated with conventional instructions still require multiple cycles to complete execution; however, the method used to encode how the operation is processed while executing differs. In SP scheduled code, the execution of the `load` operation is not specified by a single instruction. Instead each SP instruction specifies how all portions of the processor are controlled during the cycle it is executed. Initially encoding any conventional instruction may take as many SP instructions as the number of pipeline stages in a conventional processor. While this approach may seem inefficient in specifying the functionality of any single conventional instruction, the cost is offset by the fact that multiple SP effects can be scheduled for the same SP instruction. The SP instruction set architecture (ISA) results in more control given to the compiler to optimize instruction flow through the processor, while simplifying the hardware required to support hazard detection, data forwarding, and control flow. By relying on the compiler to do low-level processor resource scheduling, it is possible to eliminate some structures (e.g., the branch target buffer), avoid some repetitive computation (e.g., sign extensions and branch target address calculations), and greatly reduce accesses to both the register file and internal registers. This strategy results in improved energy efficiency through simpler hardware, while providing new code optimization opportunities for achieving performance improvement. The cost of this approach is the additional complexity of code generation and compiler optimizations targeting an SP architecture. The novelty of this paper is not strictly in the SP architectural features or compiler optimizations when either is viewed in isolation, but also in the SP ISA that enables the compiler to more finely control the processor to produce a more energy efficient system.

This paper makes the following contributions. (1) We show that the use of static pipelining can expose many new opportunities for compiler optimizations to avoid redundant or unnecessary operations performed in conventional pipelines. (2) We establish that a very low-level ISA can be used and still achieve performance and code size improvements. (3) We demonstrate that static pipelining can reduce energy usage by significantly decreasing the number of register file accesses, internal register writes, branch predictions, and branch target address calculations, as well as completely eliminating the need for a branch target buffer. In summary, we provide a novel method for pipelining that exposes more control of processor resources to the compiler to significantly improve energy efficiency while obtaining counter-intuitive improvements in performance and code size.

2. Architecture

In this section we discuss the SP architecture design including both the micro-architecture and the instruction set. While static pipelining refers to a class of architectures, we describe one design in detail that is used as the basis for the remainder of this paper.

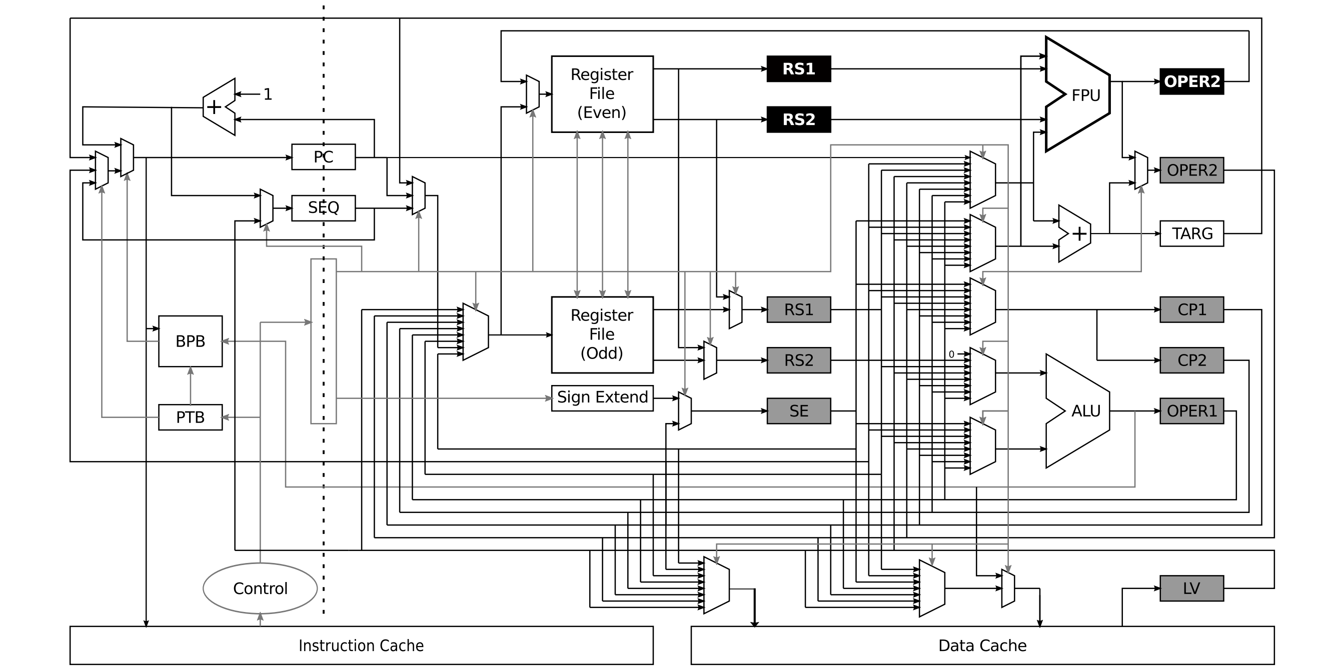

Figure 3. Datapath of a Statically Pipelined Processor

2.1 Micro-Architecture

The SP micro-architecture evaluated in this paper is designed to be similar to a classical five-stage pipeline in terms of available hardware and operation. The motivation for this design is to minimize the required hardware resource differences between the classical baseline design and our SP design in order to evaluate the benefits of the SP technique.

Figure 2 depicts a classical five-stage pipeline. Instructions spend one cycle in each stage of the pipeline, which are separated by pipeline registers. Along with increased performance, pipelining introduces a few inefficiencies into a processor. First of all is the need to latch information between pipeline stages. All of the possible control signals and data values needed for an instruction are passed through the pipeline registers to the stage that uses them. For many instructions, much of this information is not used. For example, the program counter (PC) is typically passed through the pipeline for all instructions, but is only used for branches.

Pipelining also introduces branch and data hazards. Branch hazards occur because the branch target address is unknown for multiple cycles after fetching a branch. These hazards result in either a pipeline flush for every taken branch, or the need for a branch prediction buffer (BPB), branch target buffer (BTB), and delays when branches are mis-predicted. Data hazards occur when the value produced by an earlier instruction is required before the pipeline has written it back to the register file. Data hazards can be eliminated in most cases with forwarding logic, which contributes to the energy usage of the pipeline. In addition, if the value is only ever consumed by instructions via forwarding paths, the value will be unnecessarily written to the register file at the commit stage, wasting energy. We performed preliminary experiments with SimpleScalar [1] running the MiBench benchmark suite [10] that indicate that 27.9% of register file reads are unnecessary because the values will be replaced from forwarding. Additionally 11.1% of register file writes are not needed due to their only consumer instructions getting the values from forwarding instead. Additional inefficiencies found in traditional pipelines include repeatedly calculating invariant branch target addresses and adding an offset to a register to form a memory address even when that offset is zero.

Figure 3 depicts one possible datapath of an SP processor. The fetch portion of the processor is mostly unchanged from the conventional processor. Instructions are fetched from the instruction cache and branches are predicted by a BPB. One difference is that there is no longer any need for a BTB. This structure is used to store the target addresses of branches in conventional pipelines, avoiding the need to wait for the target address calculation to begin fetching the next instruction when that branch is predicted to be taken. In SP, the branch target address calculation is decoupled from the transfer of control (ToC), which eliminates the need for a BTB since the address is available when the branch target is fetched. In addition, the instruction prior to a ToC sets the `PTB` (prepare-to-branch) status register to provide information about the ToC, which enables the BPB to be only accessed for conditional branches.

There are more substantial differences in the processor after instructions are fetched. There is no need for pipeline registers because SP processors do not need to break instructions into multiple stages. In their place are a number of architecturally visible internal registers. Unlike pipeline registers, these internal registers are explicitly read and written by the instructions, and can hold their values across multiple cycles.

There are ten internal registers in our SP design. The `RS1` and `RS2` (register source) registers contain values read from the register file. The `LV` (load value) register is assigned a value loaded from the data cache. The `SE` (sign extend) register receives a sign-extended immediate value. The `OPER1` (ALU result) register is updated with values calculated in the ALU. The `OPER2` (FPU result) register acquires results calculated in the FPU, which is used for multi-cycle operations, and integer addition results. The `TARG` (target address) register takes the result of adding the program counter (PC) and the `SE`. The `SEQ` (sequential address) register gets the address of the next sequential instruction at the time it is written. The `CP1` and `CP2` (copy) registers hold values copied from one of the other internal registers.

Since these internal registers are small, can be placed near the portion of the processor that accesses them, and are explicitly accessed, each internal register is accessible at a lower energy cost than the centralized register file. Note that while the pipeline registers of the baseline processor are read and written every cycle, the SP internal registers are only updated when needed. Because these internal registers are exposed at the architectural level, a new level of compiler optimizations can be exploited as we will demonstrate in Section 3.

All of the internal registers are caller save (*scratch*) registers, except for `SEQ`, `CP1` and `CP2`. These three internal registers are callee save because our optimizing compiler primarily uses them to perform aggressive loop optimizations. If a loop has a function call in it, the compiler would disallow the use of these registers for this optimization were they caller save.

Because the internal registers are part of the machine state, they must be saved and restored together with the register file upon context switches and interrupts to allow for precise exceptions. Thus, each internal register must be able to be stored to, and loaded from, memory. Some of these registers have a direct path to/from memory, while others must first be moved through a copy register or the register file.

The integer and floating-point register files are merged into a single 32 entry register file for the SP datapath as the ALU operations are decoupled from accessing the register file. In the SP datapath explicit support is shown for dealing with double-precision values, which requires having extra components shown in black for the `RS1`, `RS2`, and `OPER2` internal registers.

Data hazards due to multi-cycle operations can easily be detected without special logic to compare register numbers obtained from instructions. If during a given cycle the `OPER2` register is to be used as a source and the FPU has not completed a multi-cycle operation, then the current instruction is aborted and the instruction will be reattempted on the next cycle. This process continues until the FPU has completed the operation. Data cache misses can be handled in a similar fashion for `LV` register reads.

An SP can be viewed as a two-stage processor with the two stages being fetch and everything after fetch. As discussed in the next subsection, SP instructions are already partially decoded as compared to traditional instructions.

2.2 Instruction Set Architecture

The instruction set architecture (ISA) for an SP architecture is quite different than the ISA for a conventional processor. Each instruction consists of a set of effects, each of which updates some portion of the processor. The effects mostly correspond to what the baseline classical five-stage pipeline can do in one cycle, which includes one ALU operation, one FPU operation, one data cache access, two register reads, one register write, and one sign extension. In addition, one copy can be made from an internal register to one of the two copy registers and the next sequential instruction address can be saved in the `SEQ` register. Lastly, the `PTB` status register can be set to indicate that the next instruction is a ToC.

All of the effects specified in a single instruction are independent and are performed in parallel. The values in the internal registers are read at the beginning of the cycle and written at the end of the cycle. Note that except for the effects that solely read or write a register file or data cache value, all of the effects operate solely on the internal registers. This is analogous to how RISC architectures only allow load or store instructions to reference memory locations.

Including all possible instruction-effect fields in an instruction would require 77 bits for our design. More than doubling the size of each instruction would have a very negative effect on code size, as well as increasing the power to access the instruction cache, which would negate much of the power benefit static pipelining would otherwise achieve. Therefore we developed a compact, 32-bit encoding for the instructions, which is shown in Figure 4.

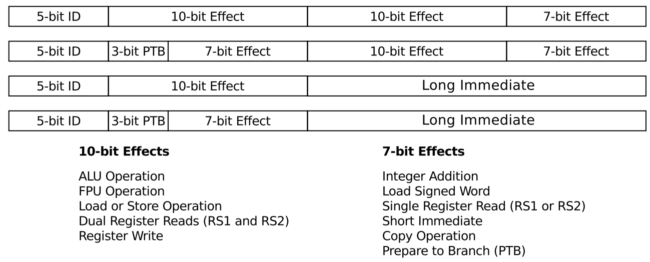

Figure 4. Static Pipeline Instruction Formats

The encoding scheme is similar to that used by many VLIW processors that use longer instruction formats. Each instruction is capable of encoding a number of fields, with each field corresponding to one SP effect. The *5-bit ID* field is the template identifier, which specifies how the remaining fields should be interpreted. The size of this identifier dictates how many combinations of fields the encoding supports. With a larger number of combinations, there is more flexibility in scheduling, but the template identifier would require more space and the decoding logic would be more complicated. Frequently used effects, such as an ALU operation, should be present in more combinations than less frequently used effects, such as copying an internal register to a copy register. Each type of field has to be present in at least one combination, or it would be impossible to use it. Figure 4 also shows which types of fields can be represented in the different size effects. Most of these fields also have a representation to indicate that no effect associated with that field is to be performed.

The templates are constructed such that each type of effect only appears in at most two distinct places across all instructions, which greatly simplifies the decoding logic. Depending on the timing breakdown of the fetch and execute cycles, this logic can either be at the end of fetch or split between the fetch and execute stage. If necessary, then we can add a stage for decoding instructions, which is discussed in more detail in Section 6. Note that, unlike a conventional five-stage RISC architecture, register file reads occur during the execute stage, so there is no need for a decode stage to fetch values from the register file.

There are many possible ways to combine the available instruction effects into a set of templates for the encoding. In order to choose a good set of templates, we compiled and simulated the MiBench benchmark suite [10] with the ability to use any combination of effects whether it could fit in 32 bits or not in order to determine which combinations of instruction effects were commonly used together. We used this information to guide our choosing of the templates. The 32 templates chosen are able to cover over 81.7% of the combinations used when no restrictions are in place. The compiler makes use of the set of selected templates in order to schedule legal combinations of effects in instructions.

3. Compilation

In this section, we will describe the compilation process in more detail and show how example code can be compiled efficiently for an SP processor. For an SP architecture, the compiler is responsible for controlling each part of the datapath for every cycle, so effective compilation optimizations are necessary to achieve acceptable performance and code size goals. Because the instruction set architecture 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.

3.1 Overview of the Compilation Process

We have ported the VPO compiler [2] to the SP processor. We believe the selection of the VPO compiler was a good choice as it uses Register Transfer Lists (RTLs) for its intermediate representation, which is at the level of machine instructions. A low level representation is needed for performing code improving transformations on SP generated code.

Figure 5 shows the steps of our compilation process. First, C code is input to the frontend, which consists of the LCC compiler [9] frontend combined with a *middleware* process that converts LCC's output format into the RTL format used by VPO.

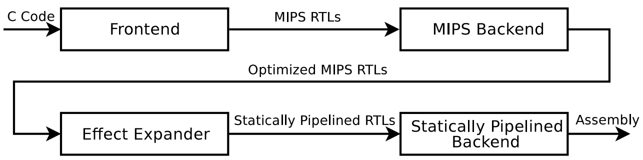

Figure 5. Compilation Process

These RTLs are then input into a modified VPO MIPS backend, which performs many compiler optimizations including control flow optimizations, loop invariant code motion, register allocation, and data flow optimizations. These optimizations are performed before converting the instructions to those for the SP architecture because some of these optimizations are more difficult to apply on the lower level instruction-pipeline representation, which breaks many assumptions in a compiler backend. For instance, register allocation is difficult to perform directly on SP instructions due to the need to have either `RS1` or `RS2` available to load any registers and that the address specifications and memory references are decoupled. Additionally this strategy allows us to concentrate on optimizations specific to the SP as all higher level (conventional) optimizations are already performed. Additional changes to the MIPS backend include using a single register file for general-purpose and floating-point values and changing some of the calling conventions. A single register file is used since the SP instructions separate accessing the register file from the ALU and FPU operations.

The instructions are next broken down into SP instruction effects by the *effect expander*, which 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 with a single effect per RTL that together perform the same computation.

Lastly, these instructions are fed into the SP compiler backend, also based on VPO. This backend applies additional optimizations and produces the final assembly code.

3.2 Example and More Detailed Discussion

In this section we describe an actual example of how code is translated to SP instructions and more thoroughly optimized by our compiler. This example illustrates that the role of the compiler is greater for an SP architecture.

Figure 6(a) shows the C source code of a simple loop kernel to add a value to every element of an array. The first step in compiling this code for the SP is to generate the optimized MIPS code which can be seen in Figure 6(b). Here `r[9]` is used as a pointer to the current element of the array, `r[6]` holds the value of the loop-invariant `m` variable, and `r[5]` has had the value `a + 400` loaded into it which will equal the last value assigned to `r[9]` in the loop. There are no other obvious optimizations to this MIPS code that can be applied without increasing the code size. Figure 6(c) shows the requirements for processing each element of the array when compiled for this MIPS code.

Figures 6(d) through 6(k) depict the actual compiler optimizations that are performed on the SP instructions for this example. To better illustrate the changes associated with each optimization, effects that are to be updated are shown in italics, effects that are to be removed are displayed with lines through them, and effects that were updated are shown in bold.

Effect Expansion The next step in the compilation process is to expand the MIPS instructions into SP instructions. The result of this step can be seen in Figure 6(d). Dashed lines separate instruction effects corresponding to each of the MIPS instructions in Figure 6(b). These instruction effects correspond to the pipeline stages performed by a five-stage MIPS processor when executing the given instruction. For example, for the add instruction, the MIPS processor will first load the registers into internal registers, next perform the addition, and then write the value back to the register file. This expansion increases the loop from five MIPS instructions to nineteen SP instructions. Note the PC is implicitly incremented after each individual SP instruction, just as the PC is incremented in a classical architecture. The offset to the branch target (`L2`) represents a symbolic offset from the `TARG` assignment and the exact offset is determined by the assembler.

Copy Propagation The first optimization is copy propagation, which is an optimization that takes into account instruction effects that copy a source to a destination and creates equivalence classes among elements that have the same value. It then replaces uses of any member of the equivalence class with the oldest member of the class with the same or cheaper access cost where possible. This optimization is applied three times. The values to be replaced are depicted in italics in Figure 6(d) and the replaced values are shown in bold in Figure 6(e). For instance, the use of `RS1` is replaced with a use of `LV` in the sixth instruction of the loop. Likewise, `OPER2` replaces uses of `RS1` and `RS2`.

Dead and Redundant Assignment Elimination The copy propagation optimization is not useful on its own, but it is helpful in that it enables other data flow optimizations, such as dead assignment elimination. Dead assignments are those that write a value into a register or memory location that is never used before being overwritten. The instructions with solid lines through them in Figure 6(e) are assignments that are now dead. In the example, the dead assignments to `RS1` and `RS2` are removed first, which causes the two writes to the register file to become dead. The redundant assignment elimination optimization is also shown in the same figure. Redundant assignments are those that assign a value to a register that already has that value. Because the MIPS code reads a value from a register every time it needs it, it is common to repeatedly load the same value. When generating SP code, however, the compiler can simply retain a value in one of the internal registers. The value of `r[9]` is assigned to `RS1` three times without the values changing in between in Figure 6(e), so the compiler removes the last two of these assignments to `RS1`, which are shown with dashed lines through them.

The resulting code after removing these dead and redundant assignments is displayed in Figure 6(f). Besides representing the SP effects and their costs in the compiler, no other changes were required to perform copy propagation and dead and redundant assignment elimination, or common subexpression elimination on SP instructions. In addition to removing seven instructions in the loop, the compiler has completely eliminated the use of two registers (`r[2]` and `r[3]`) to hold intermediary values. In a traditional pipeline, all data values circulate through the centralized register file. By giving the compiler access to internal registers, static pipelining can avoid accessing the register file in many cases.

```
for (i=0; i<100; i++)
   a[i] += m;
```
(a) Source Code

```
  r[6]=<value m>;
  r[9]=<address a>;
L2:
  r[3]=M[r[9]+0];
  r[2]=r[3]+r[6];
  M[r[9]+0]=r[2];
  r[9]=r[9]+4;
  PC=r[9]!r[5],L2;
```
(b) MIPS Code

5 instructions	5 ALU opers
1 DC load	1 DC store
8 RF reads	3 RF writes
1 targ calc	2 sign extends

(c) MIPS Requirements for Processing Each Array Element

```
  r[6]=LV;
  r[9]=OPER2;
L2:
  RS1=r[9];
  LV=M[RS1];
  r[3]=LV;
  RS1=r[3];
  RS2=r[6];
  OPER2=RS1+RS2;
  r[2]=OPER2;
  RS1=r[9];
  RS2=r[2];
  M[RS1]=RS2;
  SE=4;
  RS1=r[9];
  OPER2=RS1+SE;
  r[9]=OPER2;
  SE=offset(L2);
  TARG=PC+SE;
  RS1=r[9];
  RS2=r[5];
  PC=RS1!RS2,TARG(L2);
```
(d) Expanded Statically Pipelined Instructions

```
  r[6]=LV;
  r[9]=OPER2;
L2:
  RS1=r[9];
  LV=M[RS1];
  ~~r[3]=LV;~~
  ~~RS1=r[3];~~
  RS2=r[6];
  OPER2=LV+RS2;
  ~~r[2]=OPER2;~~
  ~~RS1=r[9];~~
  ~~RS2=r[2];~~
  M[RS1]=OPER2;
  SE=4;
  ~~RS1=r[9];~~
  OPER2=RS1+SE;
  r[9]=OPER2;
  SE=offset(L2);
  TARG=PC+SE;
  ~~RS1=r[9];~~
  RS2=r[5];
  PC=OPER2!RS2,TARG(L2);
```
(e) After Copy Propagation

```
  r[6]=LV;
  r[9]=OPER2;
L2:
  RS1=r[9];
  LV=M[RS1];
  RS2=r[6];
  OPER2=LV+RS2;
  M[RS1]=OPER2;
  SE=4;
  OPER2=RS1+SE;
  r[9]=OPER2;
  SE=offset(L2);
  TARG=PC+SE;
  RS2=r[5];
  PC=OPER2!RS2,TARG(L2);
```
(f) After Dead and Redundant Assignment Elimination

```
  r[6]=LV;
  r[9]=OPER2;
  SE=offset(L2);
  TARG=PC+SE;
  SE=4;
L2:
  RS1=r[9];
  LV=M[RS1];
  RS2=r[6];
  OPER2=LV+RS2;
  M[RS1]=OPER2;
  OPER2=RS1+SE;
  r[9]=OPER2;
  RS2=r[5];
  PC=OPER2!RS2,TARG(L2);
```
(g) After Loop–Invariant Code Motion

```
  CP1=LV;
  CP2=OPER2;
  SE=offset(L2);
  TARG=PC+SE;
  SE=4;
L2:
  ~~RS1=r[9];~~
  LV=M[CP2];
  ~~RS2=r[6];~~
  OPER2=LV+CP1;
  M[CP2]=OPER2;
  OPER2=CP2+SE;
  CP2=OPER2;
  RS2=r[5];
  PC=OPER2!RS2,TARG(L2);
```
(h) After CP Register Allocation

```
  CP1=LV;
  CP2=OPER2;
  SE=offset(L2);
  TARG=PC+SE;
  SE=4;
  RS2=r[5];
L2:
  LV=M[CP2];
  OPER2=LV+CP1;
  M[CP2]=OPER2;
  OPER2=CP2+SE;
  CP2=OPER2;
  PC=OPER2!RS2,TARG(L2);
```
(i) After Dead Asg Elimination and Loop–Invariant Code Motion

```
  CP1=LV;
  CP2=OPER2;
  ~~SE=offset(L2);~~
  ~~TARG=PC+SE;~~
  SE=4;
  RS2=r[5];
  SEQ=PC+1;
L2:
  LV=M[CP2];
  OPER2=LV+CP1;
  M[CP2]=OPER2;
  OPER2=CP2+SE;
  CP2=OPER2;
  PC=OPER2!RS2,SEQ(L2);
```
(j) After Using SEQ Register

```
  CP1=LV;          SE=4;             RS2=r[5];
  CP2=OPER2;       SEQ=PC+1;
L2:
  LV=M[CP2];       OPER2=CP2+SE;
  OPER1=LV+CP1;    PTB=b:SEQ(L2);
  M[CP2]=OPER1;    CP2=OPER2;        PC=OPER2!RS2,SEQ(L2);
```
(k) After Dead Assignment Elimination and Effect Scheduling

3 instructions	3 ALU opers
1 DC load	1 DC store
0 RF reads	0 RF writes
0 targ calc	0 sign extends

(l) SP Requirements for Processing Each Array Element

Figure 6. Compilation Example

Loop-Invariant Code Motion The next optimization shown is loop-invariant code motion, which moves loop-invariant values to the preheader block preceding of the loop. The italic instructions in Figure 6(f) are loop invariant and are hoisted out of the loop by our compiler. Because the branch target address does not change, the two instructions that calculate the address are first moved out of the loop. Hoisting branch target address calculations out of a loop required a new machine-specific optimization since the two effects have to update the SE and TARG registers. At this point there is only one remaining assignment to SE, so this sign extension is also moved out of the loop. Figure 6(g) shows the result of hoisting these instructions out of the loop, which improves both performance and energy usage. With traditional architectures, these computations are loop invariant, but cannot be moved out with compiler optimizations due to the fact that these computations cannot be decoupled from the instructions that use them.

CP Register Allocation Similar to allocating live ranges of scalar variable references to registers, our SP backend assigns live ranges of register file references to CP registers. In Figure 6(g) there are live ranges of r[6] and r[9] that span both the preheader and the loop. The compiler connects each live range of a register with the live range of the RS register into which its value is loaded. The compiler assigns an available CP register if the CP register is not live in the connected live range. Figure 6(h) shows the assignments to r[6] and r[9] replaced with CP1 and CP2, respectively. Likewise, the uses of RS1 and RS2 loaded from these registers are also replaced. Note that internal register accesses, such as CP1, require less energy than a register file access. At this point the loads from r[6] and r[9] are dead assignments and are eliminated, as shown in Figure 6(i). Thus, CP register allocation not only replaces register file references with internal register references, but also reduces the number of effects. Figures 6(h) and 6(i) also show that there is only one remaining assignment to RS2 and the compiler hoists it out of the loop by applying loop-invariant code motion.

The CP register allocation optimization requires careful heuristics as there are only two CP registers available in our design. We not only have to estimate the number of register load effects that would be eliminated by allocating a live range, but also the number of effects that cannot be eliminated in conflicting live ranges due to allocating the current live range. In addition, there is a definitive cost of using CP registers due to the need to save and restore these callee-save registers and eliminating a register load effect does not necessarily decrease the number of instructions after instruction scheduling. Note that we initially implemented optimizations to hoist register file references out of only the innermost loops of a function by using CP registers. However, we found that we not only missed opportunities for eliminating register file references outside of these loops, but also did not make the best use of CP registers within these loops by not using live range analysis.

SEQ Register The next optimization shown is using the SEQ register to store the target address of the loop branch to eliminate the calculation of the address. The instruction that saves the next sequential address at the start of the loop is inserted, and the loop branch is modified to jump to SEQ instead of TARG. The result after this machine-dependent transformation can be seen in Figure 6(j). The two instructions that calculate the value of TARG are then removed by the dead assignment elimination optimization. The SEQ optimization also allows a second branch target address calculation to be hoisted out of a loop. Note that TARG(L2) or SEQ(L2) simply indicate that the target address comes from the internal register and the label in parentheses depicts that the target address is known to the compiler.

Effect Scheduling The optimizations to this point have reduced the original 19 instructions in the loop to only six. In order to further reduce the number of instructions, we schedule multiple instruction effects together in parallel. Figure 6(k) shows the code after scheduling is applied. The SP instruction scheduler must respect structural hazards as well as dependencies between instructions. Because of the fact that the internal registers are used so frequently, and each has a prescribed purpose, code compiled for the SP architecture typically has far more anti-dependencies than code for other machines. As a part of scheduling, we attempt to rename some internal registers to avoid these anti-dependences. In the figure the compiler renames the result of the first addition in Figure 6(j) to use OPER1 instead of OPER2, as both registers can be assigned an integer addition result.

Although not shown in this example, our scheduler also moves instruction effects across basic blocks to obtain greater performance improvements and code size reductions. We attempt to move instruction effects into each of its predecessor blocks if there is an available slot in an instruction and moving the effect does not violate any data dependencies, the effect cannot cause a fault (e.g., load or store), and the effect is not considered too expensive (e.g., multiply or divide).

Handling Branches The compiler also splits the branch effect into two effects in Figure 6(k). First, the PTB status register is set to specify the type and where the branch target address is to be obtained. In the next instruction, the comparison is then performed and the actual transfer of control takes place. As discussed in Section 2, the presence of a branch is specified one instruction ahead of time to avoid performing branch predictions on every instruction. This strategy also completely eliminates the need for a BTB as target addresses are calculated before transfers of control, which are explicitly identified in the preceding instruction.

Resource Utilization Figure 6(l) shows the pipeline requirements for each element of the array in the code produced by our SP compiler. The SP code has three instructions in the loop as compared to five for the MIPS code. All accesses to the register file inside the loop for the SP code have been eliminated. The SP code also reduced the number of ALU operations by not adding zero when calculating a memory address, and eliminated sign extensions and branch target address calculations.

Immediate Transformation There are other optimizations that our compiler performs that are not illustrated in Figure 6. One such optimization is to transform large constants (immediates) to small constants when possible. As shown in Figure 4, large constants require 17 bits in our encoding and are used to load a 16-bit value into the low or high portion of the SE register. If a small constant that fits in our 7-bit field can instead be used, then additional effects may be placed in the instruction. Figure 7(a) shows a transformation that can sometimes accomplish this goal. Assume that *const1* and *const2* both require a large immediate field, but the difference between the two constants can fit in the small immediate field. Likewise, assume that OPER2 and RS1 are not updated between the two pairs of instructions shown in the figure. The compiler changes the second pair of instructions to use the difference between the two constants and replaces RS1 with OPER2. Figures 7(b) and 7(c) show instructions in the prologue of a function to save register values after scheduling without and with applying this transformation. The number of instructions is reduced from 11 to eight and four assignments to SE are also eliminated due to the remaining differences being identical.

```
SE=const1;                    SE=const1;
OPER2=RS1+SE;                 OPER2=RS1+SE;
 ...               ➡           ...
SE=const2;                    SE=const2-const1;
OPER2=RS1+SE;                 OPER2=OPER2+SE;
```

(a) General Tranformation

```
SE=188;     RS1=r[29];       RS2=r[23];
SE=184;     OPER2=RS1+SE;
            OPER2=RS1+SE;    M[OPER2]=SEQ;
SE=180;                      M[OPER2]=CP1;
SE=176;     OPER2=RS1+SE;
            OPER2=RS1+SE;    M[OPER2]=CP2;
SE=172;                      M[OPER2]=RS2;
SE=168;     OPER2=RS1+SE;
RS2=r[24];
RS2=r[25];  OPER2=RS1+SE;    M[OPER2]=RS2;
                             M[OPER2]=RS2;
```

(b) Example without the Transformation

```
SE=188;     RS1=r[29];       RS2=r[23];
SE=-4;      OPER2=RS1+SE;
            OPER2=OPER2+SE;  M[OPER2]=SEQ;
            OPER2=OPER2+SE;  M[OPER2]=CP1;
            OPER2=OPER2+SE;  M[OPER2]=CP2;
RS2=r[24];  OPER2=OPER2+SE;  M[OPER2]=RS2;
RS2=r[25];  OPER2=OPER2+SE;  M[OPER2]=RS2;
                             M[OPER2]=RS2;
```

(c) Example with the Transformation

Figure 7. Avoiding the Use of Large Constants

The compiler also encodes branch displacements as constants assigned to the SE register. These displacements can decrease during the scheduling of effects. The compiler marks each branch displacement effect that was initially scheduled into large immediate fields and reschedules a basic block if it later finds that the effect can be represented using a small immediate field. This process is iteratively performed until no such occurrences are found.

Summary of Compiler Modifications The modifications to the compiler were extensive and included expanding MIPS instructions to SP instructions, representing SP instructions in the compiler backend, loop-invariant code motion of register file reads using a CP register, loop-invariant code motion of branch target address calculations, use of the SEQ register to hold a branch target address, allocating live ranges of register file references to CP registers, transforming large immediates to small immediates, an internal register renaming pass to eliminate false dependences, placement of the PTB effect, scheduling of SP effects both within and across basic blocks. Several of these optimizations had to ensure that each transformation was legal before it could be committed due to the restricted SP datapath.

4. Evaluation

This section presents an experimental evaluation of the SP architecture including a description of the experimental setup and results for performance, code size, and an estimation of the energy savings achieved by static pipelining.

4.1 Experimental Setup

We use 17 benchmarks shown in Table 1 from the MiBench benchmark suite [10], which is a representative set of embedded applications. We extended the GNU assembler to assemble SP instructions and implemented a simulator based on the SimpleScalar in-order MIPS [1]. In order to avoid having to compile all of the standard C library and system code, we allow SP code to call functions compiled for the MIPS. A status bit is used to indicate whether it is a MIPS or SP instruction. After fetching an instruction, the simulator checks this bit and handles the instruction accordingly. On a mode change, the simulator will also drain the pipeline.

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

Table 1. Benchmarks Used

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. Were all the library code compiled for the SP as well, the results would likely improve as we would not need to flush on a mode change, and we would also have the energy saving benefits applied to more of the 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.

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.

Each bar represents a different benchmark except for the averages. The ratios are averaged rather than the raw numbers to weight each benchmark evenly rather than giving greater weight to those that run longer. When a given benchmark had more than one simulation associated with it (e.g., jpeg has both encode and decode), we averaged the figures for all of its simulations and then used that figure for the benchmark to avoid weighing benchmarks with multiple runs more heavily.

4.2 Results

Figure 8 shows the ratios for simulated execution cycles. Many of the benchmarks in MiBench are dominated by fairly tight loops. This means that the performance difference is largely determined by how well the SP compiler does on these kernel loops. That is the primary reason for the relatively large deviation among benchmarks. For example, our compiler does quite well with the kernel loops in *dijkstra*, *qsort*, *sha*, and *stringsearch* which leads to the substantial speedups. On the other hand, *adpcm*, *bitcount*, and *ispell* have more control flow in their kernel loops, leading to execution time increases due to the previously mentioned restrictions of scheduling effects across basic blocks. On average, the SP code performed 7.9% better than the MIPS code. This figure also shows that not enforcing the 32-bit instruction size restriction results in 6.5% fewer cycles as compared to using templates, as described in Section 2.2. We describe in Section 6 how we may be able to avoid some of this performance degradation from using templates without significant increases in code size.

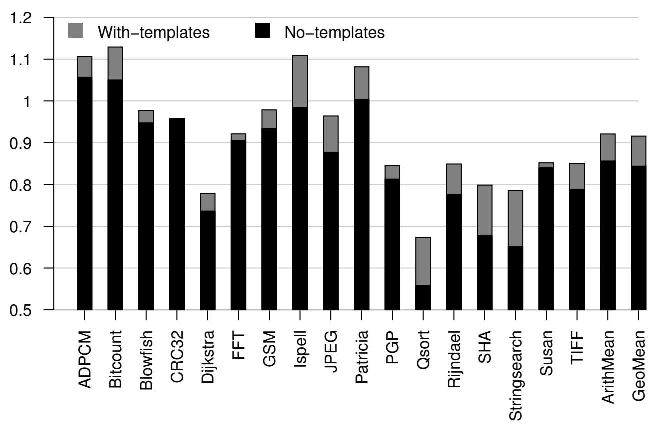

Figure 8. Execution Cycles

Figure 9 shows the compiled code size ratios for the benchmarks. The SP compiler produces code size that is 8.3% smaller than the MIPS compiler on average. These performance and code size improvements are counter-intuitive given that a lower level instruction format is used, but is due to eliminating many SP effects.

Table 2 summarizes the average (arithmetic mean) results from the simulations. Because the SP is able to use values in internal registers directly, it is often able to bypass the centralized register file as discussed in Section 3. For this reason, we are able to remove 74% of the register file reads. For the MIPS baseline pipeline, we only count register reads when the instruction actually references the register file, which is not the case for some pipeline implementations. Like register reads, the compiler is able to remove a substantial number of register file writes, 67% on average. As depicted in the example in Section 3.2, some loops had nearly all of the register accesses removed, such as *rijndael* and *CRC32*. Because the register file is a fairly large structure that is frequently accessed, these register access reductions should result in substantial energy savings. For the MIPS programs, *internal writes* are the number of

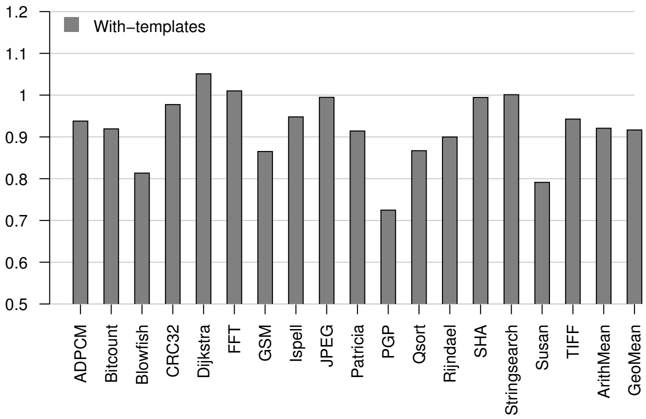

Figure 9. Code Size

Metric	Average SP to MIPS Ratio
Execution Cycles	0.92
Code Size	0.92
Register File Reads	0.26
Register File Writes	0.33
Internal Writes	0.39
Branch Predictions	0.13
Target Calculations	0.59
BTB Accesses	0.00

Table 2. Summary of Results

writes to the pipeline registers. We evaluate each pipeline register as a single element, even though the components of these registers can be viewed separately, as shown in Figure 2. Because there are four such registers, and they are written every cycle, this figure is simply the number of cycles multiplied by four. For the SP, the *internal writes* refer to writes to the internal registers. Because the SP code explicitly instructs the architecture when to write an internal register, we are able to remove 61% of these writes, on average. The SP specifies when a conditional branch will occur one cycle ahead of time, which eliminates the need to predict branches except when the instruction actually is a conditional branch. This results in an 87% average decrease in the number of branch prediction buffer accesses. Because the SP has the ability to avoid calculating branch targets for innermost loops by saving the next sequential address at the top of the loop, and by hoisting these invariant branch target address calculations out of loops, we are able to substantially reduce the number of branch target calculations by 39%. In summary, we have significantly reduced the number of register file accesses, internal register accesses, branch predictions, and branch target address calculations and have completely eliminated the BTB. At the same time, we have also decreased both the number of execution cycles and code size.

4.3 Processor Energy Estimation

This section presents an estimate of the processor energy savings achieved by the SP approach. This estimate uses 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 modelled using CACTI [17]. 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 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 3.

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

Table 3. Pipeline Component Relative Power

Figure 10 shows the results of this analysis. On average, the SP reduces energy usage by 27%. 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.

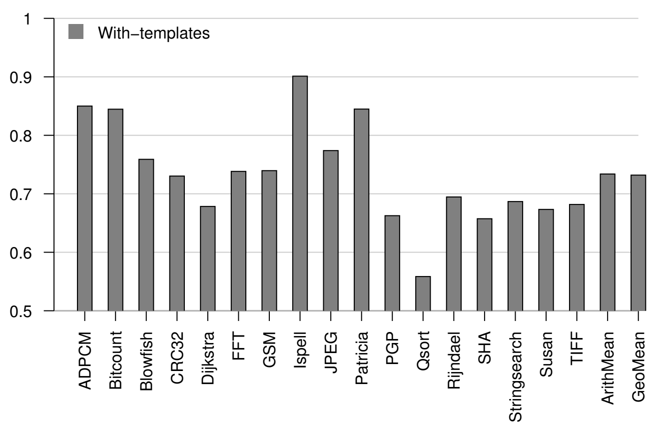

Figure 10. Estimated Energy Usage

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. The interconnect of the SP has more links than that of a classical five-stage pipeline, which could negatively effect the power and performance. The interconnect shown in Figure 3 has 69 links. Out of these, 16 links are used less than 1% of the time and could be removed with low impact to the SP's performance [12]. As discussed in Section 6, we intend to more accurately estimate energy benefits by doing full pipeline netlist simulations for both the MIPS and SP.

5. Related Work

This paper builds on previous work [6, 7] where the majority of the presented optimizations had not been implemented and only a couple of toy examples were evaluated by hand. We now have implemented all of the presented optimizations and have automatically evaluated the results on a large set of benchmark applications. Our current paper also has a number of new compiler optimizations, an updated architecture, and a completely new instruction set encoding, as described in Section 2.2.

SP instructions are most similar to horizontal microinstructions [20], however, there are significant differences. Firstly, the effects in SP instructions specify how to pipeline traditional operations across multiple cycles. While horizontal micro-instructions also specify computation at a low level, they do not expose pipelining at the architectural level. Also, in a micro-programmed processor, each machine instruction causes the execution of micro-instructions 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. It has been proposed to break floating-point operations into micro-operations and optimize the resulting code [5]. However, this approach can result in a significant increase in code size. Static pipelining also bears some resemblance to VLIW [8] 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 has previously required an entire instruction and thus may impact code size and performance. In contrast, our PTB field only requires 3 bits as the target address calculation is decoupled from both the PTB field and the point of the transfer of control.

There have been other proposed architectures that also expose much of the datapath to a compiler. One architecture that gives the compiler direct control of the micro-architecture is the No Instruction Set Computer (NISC) [14]. Unlike other architectures, there is no fixed ISA that bridges the compiler with the hardware. Instead, the compiler generates control signals for the datapath directly. The FlexCore processor [18] 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. Likewise, the TTA compiler was able to perform copy propagation and dead assignment elimination on register references. Thus, both the TTA and the SP avoid many unnecessary register file accesses. However, the SP backend performs many other optimizations that are not performed for the TTA (and the NISC and FlexCore), while using fewer internal registers. These additional optimizations include performing loop-invariant code motion of register file accesses and target address calculations, allocating live ranges of registers to internal registers, using a `SEQ` register to avoid target address calculations at the top of a loop, and transforming large immediates to small immediates. 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. An alternative TTA design did achieve comparable code size and performance compared to a RISC baseline, but required an intermixture of 16-bit and 32-bit instructions and the use of internal register queues, which increase the hardware complexity [11]. 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.

There have also been many studies that focused on increasing the energy-efficiency of pipelines by avoiding unnecessary computations. One study presents many methods for reducing the power consumption of register file accesses [19]. One method, bypass skip, avoids reading operands from the register file when the result would come from forwarding anyway. Another method is read caching, which is based on the observation that subsequent instructions will often read the same registers. Another technique that avoids unnecessary register accesses is static strands [15], where a strand is a sequence of instructions that has some number of inputs and only one output. The key idea is that if a strand is treated as one instruction, then the intermediate results do not need to be written to the register file. Strands are dispatched as a single instruction where they are executed on a multi-cycle ALU that cycles its outputs back to its inputs. All of these techniques attempt to make processors using traditional instruction sets more efficient. An SP processor avoids all of these unnecessary register file accesses without the need for special hardware logic to detect these opportunities, which can negate some of the energy savings.

6. Future Work

As discussed in Section 4, our current energy savings results are only estimates. While our results were estimated conservatively, and are still significant, it would increase the strength of this work to have more accurate results. Our current estimates are based on counting the number of times different events happen in the micro-architecture and estimating the energy costs of each event. This method does not allow us to take into account other changes in energy usage such as the fact that we no longer need to do forwarding and that hazard detection is much simpler. The SP design also includes a number of multiplexers not found in the traditional pipeline. In order to evaluate the changes in energy usage and timing of these components, we plan to construct a netlist implementation using VHDL. Because each portion of the datapath is explicitly controlled, there is less complexity in the operation of the micro-architecture. The logic for checking for hazards is much simpler, forwarding does not take place, and values are not implicitly copied through pipeline registers each cycle. Due to these factors, SP hardware should have decreased area and cost compared to equivalent traditionally pipelined hardware.

The software pipelining compiler optimization could be applied to further improve the performance of SP code. This optimization is a technique used to exploit instruction-level parallelism in loops [16]. Loops whose iterations operate on independent values, typically in arrays, provide opportunities for increased parallelism. Software pipelining overlaps the execution of multiple iterations and schedules instructions in order to allow the micro-architecture to take advantage of this parallelism. Software pipelining would have little benefit for the baseline MIPS, except when long latency operations, such as multiply and divide, are used. However, for an SP machine, software pipelining could be applied in order to schedule many innermost loops more efficiently. Software pipelining, however, can also have a negative effect on code size.

We encode SP instructions in order to attain reasonable code size, however this does have a negative impact on performance as compared to using a larger instruction format. In order to address these conflicting requirements, we could allow both 32-bit and 64-bit instructions in different situations. Like the *Thumb2* instruction set that supports intermixing 16-bit and 32-bit instructions [13], we could use 64-bit instructions where a higher number of effects can be scheduled and 32-bit instructions elsewhere to retain most of the code size benefits of the smaller instructions.

The design of a high performance, SP processor would likely include more internal registers, along with more functional units, and possibly more ports to the register file. This would mean that the instructions would have additional different types of effects, possibly leading to an issue with code size, though larger code sizes are generally less of an issue with general-purpose processors than with embedded ones.

7. Conclusions

Static pipelining is designed to explore the extreme of energy efficient architectural design. It utilizes a fairly radical and counter-intuitive approach for representing instructions to provide greater control of pipeline operation. The primary question about this design is if a compiler can generate code that is competitive with a more conventional representation. The challenges in this research included using a low-level representation that violated many assumptions in a conventional compiler, ensuring that transformations resulted in legal instructions given the restricted datapath, and in applying instruction scheduling to such a different target architecture. It was initially unclear how efficiently we could populate pipeline resources around control-flow instructions and if it would be possible to utilize a 32-bit format for SP instructions. Both of these challenges were resolved in our compiler.

Our SP target architecture achieves on average better performance and code size as compared to optimized code generated for the analogous conventional (MIPS) processor architecture. In quite a few cases, we were able to significantly improve performance, and the overall performance was limited by slowdowns in some benchmarks caused by idiosyncratic behavior that can be addressed with future optimizations specific to SP code. Static pipelining clearly provides a benefit from an energy perspective. By reducing accesses to pipeline (internal) registers, and eliminating unnecessary accesses to architectural and micro-architectural resources, an average energy savings of 27% is achieved. The obtained results show that it is useful to re-examine the boundary between hardware and software to improve processor energy efficiency.

8. Acknowledegements

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants CNS-0964413 and CNS-0915926, a Google Faculty Research Award, Korea SMBA grant 0004537, and KEIT 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] W. Dally. Micro-optimization of floating-point operations. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–289, 1989.

[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 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.

[8] J. Fisher. VLIW Machine: A Multiprocessor for Compiling Scientific Code. *Computer*, 17(7):45–53, 1984.

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

[10] 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.

[11] Y. He, D. She, B. Mesman, and H. Corporaal. Move-pro: A low power and high code density TTA architecture. In *International Conference on Embedded Computer Systems*, pages 294–301, July 2011.

[12] T. Hoang-Thanh, U. Jälmbrant, E. Hagopian, K. P. Subramaniyan, M. Själander, and P. Larsson-Edefors. Design Space Exploration for an Embedded Processor with Flexible Datapath Interconnect. In *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 55–62, July 2010.

[13] A. Ltd. Arm thumb-2 core technology. *http://infocenter.arm.com /help/index.jsp?topic= /com.arm.doc.dui0471c /CHDFEDDB.html*, June 2012.

[14] M. Reshadi, B. Gorjiara, and D. Gajski. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 69–76, Washington, DC, USA, 2005. IEEE Computer Society.

[15] P. Sassone, D. Wills, and G. Loh. Static Strands: Safely Collapsing Dependence Chains for Increasing Embedded Power Efficiency. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 127–136. ACM, 2005.

[16] A. Sethi and J. Ullman. Compilers: Principles Techniques and Tools. *Addision Wesley Longman*, 2000.

[17] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi. Cacti 5.1. Technical report, HP Laboratories, Palo Alto, Apr. 2008.

[18] M. Thuresson, M. Själander, 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.

[19] J. H. Tseng and K. Asanovic. Energy-efficient register access. In *SBCCI '00: Proceedings of the 13th symposium on Integrated circuits and systems design*, page 377, Washington, DC, USA, 2000. IEEE Computer Society.

[20] 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.