Scheduling Instruction Effects for a Statically Pipelined Processor

B. Davis, R. Baird, P. Gavin
Florida State University
Tallahassee, USA

M. Själander
Uppsala University
Uppsala, Sweden

I. Finlayson
University of Mary Washington
Fredericksburg, USA

F. Rasapour, G. Cook, G.-R. Uh
Boise State University
Boise, USA

D. Whalley, G. Tyson
Florida State University
Tallahassee, USA

ABSTRACT

Statically pipelined processors have a fully exposed datapath where all portions of the pipeline are directly controlled by effects within an instruction, which simplifies hardware and enables a new level of compiler optimizations. This paper describes an effect scheduling strategy to aggressively compact instructions, which has a critical impact on code size and performance. Unique scheduling challenges include more frequent name dependences and fewer renaming opportunities due to static pipeline (SP) registers being dedicated for specific operations. We also realized the SP in a hardware implementation language (VHDL) to evaluate the real energy benefits. Despite the compiler challenges, we achieve performance, code size, and energy improvements compared to a conventional MIPS processor.

Keywords

compiler, architecture, static pipeline, performance, energy

1. INTRODUCTION

Energy has now become a first-order processor design constraint, but many micro-architectural techniques were developed when energy efficiency was not as important. For instance, classical in-order scalar pipeline designs have energy inefficiencies that could potentially be addressed. They update pipeline registers often with values that are not used, access register files unnecessarily for values that will be forwarded, check for forwarding and hazards even when they cannot possibly occur, and repeatedly calculate invariant values like branch target addresses. The static pipeline (SP) architecture can often avoid these energy inefficiencies while achieving performance and code size improvements [1, 2].

Figure 1 illustrates the basic concept of SP. Conventional in-order instruction pipelines fetch each instruction and push the instruction through the pipeline via pipeline registers over several cycles; portions of the operation are performed at different stages, as depicted in Figure 1(a). Each instruction shares the pipeline with other instructions that are in other pipeline stages. Figure 1(b) shows the operation of an SP processor. The highlighted portion shows each SP instruction controls all portions of the processor during the cycle it is executed. The SP approach requires that the compiler encodes information in instructions that is normally determined dynamically by the hardware. Thus, the SP instruction set architecture (ISA) results in more control given to the compiler to optimize data flow through the processor, while simplifying the hardware required to support hazard detection, data forwarding, and transfers of control (ToCs).

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., branch target address calculations), and greatly reduce accesses to both the register file and internal registers. This strategy provides new code optimization opportunities, leading to energy efficiency, performance, and code size improvements. The cost of this approach is the additional complexity of code generation and compiler optimizations targeting an SP architecture. However, the usefulness of the SP approach is solely determined by the quality of compiler code generation. It is, therefore, vital for the compiler to detect the instruction effects that can be performed in parallel and aggressively compact these effects into the smallest possible set of SP instructions.

This paper makes the following contributions. (1) We show that a 32-bit instruction template generator can automatically select a set of encoding templates that captures the most frequent occurrences of parallel SP effects discovered by the compiler. (2) We demonstrate that SP effects that must occur within specific instructions in a basic block can be effectively scheduled with other effects within a basic block. (3) We establish that SP effects can be moved across basic blocks to provide further performance and code size improvements. (4) We show that despite frequent name dependences involving SP internal registers and the limited renaming opportunities due to the restrictions of the SP datapath, the compiler is still able to achieve a high-quality SP effects schedule. (5) We evaluate the energy efficiency of the first VHDL implementation of an SP datapath.

This paper builds on our previous work [1] where each SP effect to be scheduled was selected serially in the original order of the unscheduled effects within the basic block and register renaming was only performed as an optimization pass before effect scheduling was initiated. We now select SP effects to be scheduled using a data dependence graph (DDG), we integrate register renaming during SP effect scheduling as needed to avoid false dependences, we implement and use SP copy renaming as a new method for avoiding false dependences, we perform SP cross block scheduling much more ex-

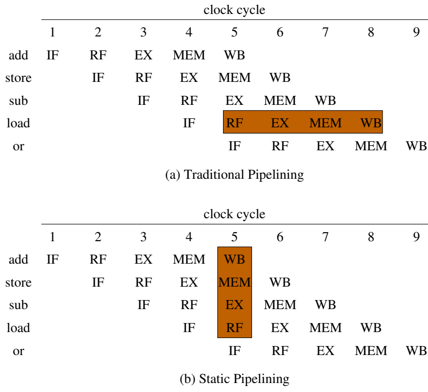

Figure 1: Traditionally Pipelined vs. Statically Pipelined Instructions

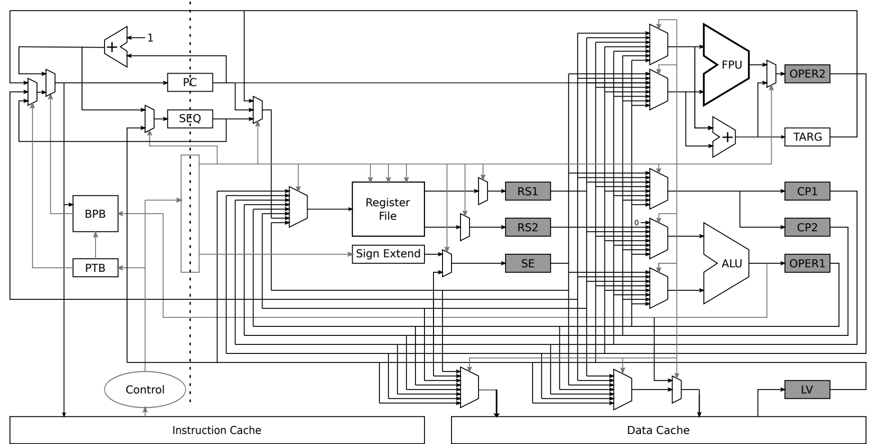

Figure 2: Datapath of a Statically Pipelined Processor

tensively that includes more aggressive techniques, we have revised the SP instruction formats, and we improved the method for selecting the SP instruction templates. We also provide an analysis of the benefits of the various scheduling techniques we implemented. Overall, the performance and code size results are about 3.2% and 2.8% better, respectively, than serially scheduled code [1]. In addition, we provide the first VHDL implementation of an SP datapath.

2. STATIC PIPELINE APPROACH

The SP ISA is quite different than the ISA for a conventional processor. An SP instruction consists of a set of independent effects, each of which controls some functionality of the processor. Internal registers are architecturally visible, which unlike pipeline registers, are explicitly read and written by the effects, and can hold their values across multiple cycles.

The SP micro-architecture used in this paper, depicted in Figure 2, has similar hardware resources as a classical five-stage pipeline. The SP effects for this configuration mostly correspond to the various operations of a classical five-stage pipeline, which includes one ALU operation (`OPER1`), one FPU operation (`OPER2`), one data cache access (`LV`), two register reads (`RS1` and `RS2`), one register write, one sign extension (`SE`), and one branch target address calculation (`TARG`). In addition, SP effects also enable a copy to be made from an internal register to one of two copy registers (`CP1` and `CP2`), to store the next sequential instruction address (`SEQ`), and to set a status register to indicate that the next instruction is a transfer of control (`PTB`).

The SP has a completely exposed datapath and the compiler is responsible for managing all routing of data and performing necessary operations on the data. Figure 3 shows a simplified example of the type of scheduling the compiler needs to perform. The first step breaks each assembly instruction into the sequence of SP effects that is required to perform the instruction, as depicted in Figure 3(a). For instance, the `add` instruction (`r[1] = r[3]+16`) consists of the following SP effects: First register *r[3]* is read, next the immediate register (`SE`) is set to the value *16*, and then the addition is performed on the two operands (this effect includes routing of the operands to the ALU). Performing an ALU operation implicitly updates the output register of the

```
# r[1] = r[3]+16;
RS1    = r[3];
SE     = 16;
OPER1  = RS1+SE;
r[1]   = OPER1;
# r[1] = r[5]-r[1];
RS1    = r[1];
RS2    = r[5];
OPER1  = RS1-RS2;
r[1]   = OPER1;
# r[3] = M[r[9]];
RS2    = r[9];
LV     = M[RS2];
r[3]   = LV;
```

(a) Initial SP Effects

```
RS1    = r[3];
SE     = 16;
OPER1  = RS1+SE;
RS2    = r[5];
OPER1  = OPER1-RS2;
r[1]   = OPER1;
RS2    = r[9];
LV     = M[RS2];
r[3]   = LV;
```

(b) Copy Prop + Dead Asg Elim

```
RS1   = r[3];        SE   = 16;     RS2 = r[9];
OPER1 = RS1+SE;      RS2  = r[5];   LV  = M[RS2];
OPER1 = OPER1-RS2;   r[3] = LV;
r[1]  = OPER1;
```

(c) SP Effect Scheduling

Figure 3: Simplified Scheduling Example

ALU (`OPER1`). Finally, the result of the addition is written back to register *r[1]*. This linear set of effects is very inefficient since each conventional machine instruction now requires three to four SP instructions.

The exposed datapath of the SP enables the compiler to statically manage data forwarding and eliminate many register reads and writes. Figure 3(b) shows that the result of the addition does not have to be written to the register file. The subtraction has instead been modified to directly use the internal register `OPER1` as one of its operands and as the subtraction also writes register *r[1]* the result of the addition does not have to be written back. The compiler also performs scheduling so that multiple SP effects can be simultaneously issued for each SP instruction, as shown in Figure 3(c). The goals of SP scheduling are to significantly reduce the execution time and decrease code size.

3. COMPILER CHALLENGES AND CODE GENERATION STRATEGY

In the following sub-sections we will cover the main challenges when scheduling efficient code for an SP processor.

3.1 Selecting Instruction Templates

Including all possible instruction-effect fields in an instruction would require 81 bits for our design. A large instruction size would have a detrimental effect on code size and increase the power to access the instruction cache, thus negating much of the power benefit an SP processor would otherwise achieve. Therefore we developed a compact, 32-bit encoding for the instructions, which is shown in Figure 4.

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

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 4: Static Pipeline Instruction Formats

The encoding scheme is similar to that used by VLIW and EPIC processors [3, 4]. Each instruction is capable of encoding multiple 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. Figure 4 also shows which types of effects can be represented in the different fields. The formats are constructed such that each type of field appears at most in two distinct places across all instructions, which simplifies the decoding logic. Furthermore, each type of effect has to be present in at least one template, or it would be impossible to use it.

There are many ways to combine the available effects into a set of templates for encoding. Figure 5 shows the process of gathering information and selecting templates. In order to choose a good set of templates, a representative set of applications is compiled. The initial compilation is performed with the only restriction being that the combination of effects has to fit one of the instruction formats as specified in Figure 4, i.e., the compiler is not limited to only 32 templates as specified by the 5-bit ID. This initial compilation using the pool of possible templates determines which combinations of effects are commonly used together. A static profile is then created from the compilation where each combination of effects representing a unique instruction is weighted based on how many times it appears in the code. Using only static profiles facilitates the template selection process as representative profile data need not be determined.

The template generator uses this profile information to automatically select the templates, using the algorithm shown in Figure 6. The 32 templates are greedily selected according to the profile information. Once a template is chosen, then all instructions in the profile that fit that template are removed before the next template is selected.

3.2 Scheduling Instruction Effects

Scheduling effects for an SP processor is similar to scheduling instructions for a VLIW processor, but has several important differences. First, SP instructions use a 32-bit instruction format that is shorter than the typical VLIW format, which introduces additional restrictions on what effects

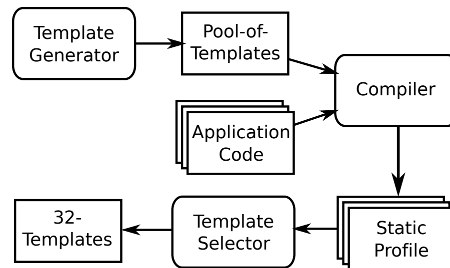

Figure 5: Process for Selecting Instruction Templates

```
P = pool of all possible 27-bit templates
Q = pool of all instructions
FOR 1..32 DO
   BEST = dummy
   BEST.freq = 0
   FOR template T in P DO
      T.freq = 0
      FOR instruction I in Q DO
         IF I can be encoded by T THEN
            T.freq += I.freq
      IF T.freq == 0 THEN
         remove T from P
      ELSE IF T.freq > BEST.freq THEN
         BEST = T
   add BEST to set of 32 templates
   remove BEST from P
   FOR instruction I in Q DO
      IF I can be encoded by BEST THEN
         remove I from Q
```

Figure 6: Algorithm for Selecting Instruction Templates

can be combined. Second, each SP effect normally represents an effect that a classical instruction pipeline can perform in a single stage. In contrast, VLIW instruction formats typically represent a complete RISC instruction. Third, many types of SP effects can only update a specified internal register, which limits internal register renaming opportunities.

The process of SP effect scheduling occurs in phases. First, the compiler schedules the effects within a basic block. Second, the compiler places specific effects that have to occur either in the last or second to last instruction within a basic block. Third, the compiler schedules effects across basic block boundaries, which requires a basic block to be rescheduled when an effect is moved out of it. The following subsections describe this process in more detail.

3.2.1 Scheduling Effects within a Basic Block

We based our scheduling on classical list scheduling, but also address the additional scheduling requirements of an SP processor. Figure 7 shows an example from scheduling SP effects within a basic block that is used to illustrate this algorithm. Figure 7(a) shows an example C `for` loop and Figure 7(b) shows the corresponding SP instructions generated for this loop prior to scheduling effects. Before scheduling the effects, the compiler removes the assignments to the `SEQ` and `PTB` registers shown in italics as these assignments have to be placed at specific locations within the block to ensure correct behavior of ToCs. The placement of the `SEQ` and `PTB` assignments after scheduling effects within a block will be described in Section 3.2.2.

The DDG the compiler uses to represent the dependences between SP effects is shown in Figure 7(c). Note that this figure only shows the true dependences (reads after writes), which depicts the constraints on the order in which the effects can be scheduled.

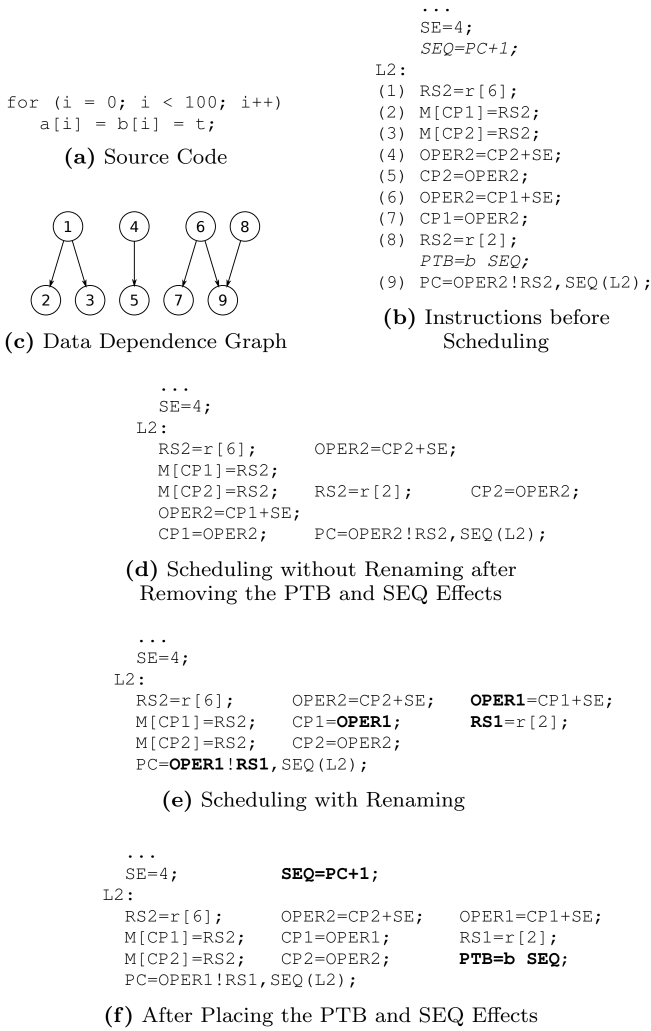

Figure 7: Example of Scheduling Effects in a Basic Block

Figure 7(d) shows the scheduled effects without performing renaming. Effects scheduled on the same line are part of the same instruction and must match a template (combination of types of effects) that both fits one of the formats shown in Figure 4 and matches a combination of effects chosen by the template selector shown in Figure 6. Effects (1) and (4) are able to be scheduled together in the first instruction, but effects (6) and (8) cannot be scheduled at this point since effects (1) and (8) set RS2 and effects (4) and (6) set OPER2. Scheduling two effects in the same instruction that set the same register would violate an output dependence. Effects (6) and (8) cannot be scheduled until the effects containing the last use of OPER2 and RS2, respectively, have been scheduled. At this point only effects (2) or (3) can be scheduled in the second instruction since OPER2, RS2, and CP2 are live. Scheduling both effects (2) and (3) in the same instruction is not legal as only one data memory operation can occur at a time and hence does not match any of the templates. After scheduling effect (2), effects (3), (8), and (5) are scheduled in the third instruction. At the point effect (5) is scheduled, effect (6) cannot fit in the third instruction, so (6) had to be scheduled in the fourth instruction. Finally, effects (7) and (9) are scheduled in the fifth instruction as they have true dependences with effect (6).

The scheduled effects in Figure 7(d) can be improved by performing renaming. Once a ready effect is selected, the scheduler attempts to rename the register set by the effect if there is only a false dependence that prevents the effect from moving higher. The conventional renaming operation performed by a compiler is when the set of a register and all uses of that value (the entire live range) are replaced with another register. However, there are only limited opportunities for conventional renaming of live ranges of internal SP registers due to the restricted SP datapath, as shown in Figure 2. There are two main opportunities for renaming internal SP registers. First, the compiler can rename a live range of RS1 to RS2. RS1 is used by default when loading the value of most register file references. RS2 is only initially used for the second register value associated with a binary operation and for a register value in an indirect jump. Second, the compiler can rename OPER2 to OPER1 when OPER2 is set by an integer addition operation.

In order to mitigate false dependencies, we rename the live range starting from the selected ready effect to be scheduled to the last use of the internal register within the remaining unscheduled effects of the block. Thus, the live range to be renamed cannot be live leaving the block being scheduled. Likewise, renaming is not performed if the effect cannot be placed higher in the set of scheduled instructions than without renaming. Note that all of the effects that reference the register to be renamed in the live range have to also be tested to ensure they are still legal for the renaming to be committed since the restricted SP datapath does not allow all internal registers to be used with all operations. Register renaming is simpler with a conventional ISA since typically a register in an instruction can always be replaced by a different register of the same type.

Figure 7(e) shows the instructions after performing renaming when scheduling effects. After effects (1) and (4) have been scheduled, the compiler is able to schedule effects (6) and (8) earlier by renaming the internal registers they set to OPER1 and RS2, respectively. So for this example performing register renaming decreases the number of instructions in the basic block from five to four.

If after internal SP register renaming there still is a false dependence preventing the SP effect from being placed higher in the set of scheduled instructions, then the compiler attempts to rename the blocking live range (already scheduled) by using a copy register. The compiler copies an internal register to CP1 or CP2 and then replaces the remaining uses in the live range of the internal register with the CP register. Renaming by copying a value is less desirable than conventional renaming as there are additional costs for copying a value. The copy requires an extra instruction effect, and the CP registers are callee-save. We currently only copy to a CP register if the CP register has already been used within the function (will already be saved and restored). Due to a lack of a good heuristic to be able to predict when the use of a copy will provide a benefit, the compiler performs the scheduling of the block twice, once without copy renaming and once with copy renaming. The version with copy renaming is only committed if it reduces the number of instructions within the block.

Figure 8 shows an example where a copy register is used to avoid a false dependence. Figure 8(a) shows instructions scheduled without performing copy renaming. The SE=12; effect cannot be moved higher in the block due to the antide-

pendence with the OPER1=OPER1>>SE; effect. Figure 8(b) shows instructions scheduled when copy renaming is performed. The extra CP1=SE; effect is placed after the point where SE is set in the scheduled instructions. Now the SE=12; effect can placed earlier, which in turn enables other effects that have dependences on SE to be placed earlier.

```
...; SE=16;
...;
...
OPER1=LV<<SE;
OPER1=OPER1>>SE;  SE=12;
OPER2=SE+RS2;     SE=4;
...
```

(a) Scheduled Instructions

```
...; SE=16;
...; CP1=SE;
...
OPER1=LV<<CP1;     SE=12;
OPER1=OPER1>>CP1;  OPER2=SE+RS2;  SE=4;
...
```

(b) Scheduled Instructions with Copy Renaming

Figure 8: Example of Scheduling Using Copy Renaming

Because the placement of the PTB effect determines when the branch will occur, and the placement of the SEQ effect determines the address that will be stored in the SEQ register, these effects must be placed at specific positions within the block. Both of these types of effects are removed before scheduling and then placed in the appropriate positions afterwords. Figure 7(f) illustrates the placement of the PTB and SEQ effects. While both the PTB and SEQ effects have no dependences with other instructions, there are still some complications that have to be addressed since there may not be space or an available template that allows these effects to be placed with an existing instruction. When the compiler cannot place the SEQ effect in the last instruction within a basic block, then it simply adds the SEQ effect as an additional instruction. When the compiler cannot place the PTB effect in the second to last instruction within a basic block, it tries to move other effects in that instruction to the last instruction in an attempt to open up a slot for the PTB effect. If the compiler is unsuccessful, then it simply inserts the PTB effect as a separate instruction before the last instruction.

3.2.2 Scheduling Effects across Basic Blocks

Our cross-block scheduling algorithm processes the blocks in the order of innermost loops first, and repeatedly processes the blocks in the function until no cross block scheduling changes can be performed. For each block the algorithm attempts to move effects into available slots of its predecessor blocks. The scheduler does not move an effect into a predecessor block if that block is at a deeper loop nesting level to avoid repetitively performing an effect whose result will only be used after the loop. After moving effects out of a block, the effects in the block are then rescheduled since the newly available empty slots may allow the number of instructions in the block to decrease. If the number of instructions in the current block increased after rescheduling the block (which can infrequently occur due to the use of a greedy scheduling algorithm), then the instructions in the block and its predecessor blocks are restored.

Figure 9 shows an example of scheduling instructions across basic blocks. Figure 9(a) shows the source of a C for loop with an if statement and Figure 9(b) depicts the corresponding SP instructions before scheduling the effects, where each of the instructions in the loop are numbered. The first block in the loop has a label of L2 and consists of effects (1) to (4), the second block in the loop consists of the single effect (5), and the third block in the loop has a label of L6 and consists of effects (6) to (9).

```
max = a[0];
for (i = 1; i < 100; i++)
   if (a[i] > max)
      max = a[i];
```

(a) Source Code

```
    ...
    SE=4;
    RS2=r[3];
    SEQ=PC+1;
   L2:
(1) LV=R[CP1];
(2) OPER1=CP2<LV;
(3) PTB=b TARG;
(4) PC=OPER1:0,TARG(L6);
(5) CP2=LV;
   L6:
(6) OPER2=CP1+SE;
(7) CP1=OPER2;
(8) PTB=b SEQ;
(9) PC=OPER2!RS2,SEQ(L2);
```

(b) Instructions before Scheduling

```
 ...              SE=4;
 RS2=r[3];        SEQ=PC+1;
L2:
 LV=R[CP1];
 OPER1=CP2<LV;    PTB=b TARG;
 PC=OPER1:0,TARG(L6);
 CP2=LV;
L6:
 OPER2=CP1+SE;    PTB=b SEQ;
 CP1=OPER2;       PC=OPER2!RS2,SEQ(L2);
```

(c) After Basic Block Scheduling

```
 ...              SE=4;
 RS2=r[3];        SEQ=PC+1;
L2:
 LV=R[CP1];       OPER2=CP1+SE;
 OPER1=CP2<LV;    PTB=b TARG;
 PC=OPER1:0,TARG(L6);
 CP2=LV;          OPER2=CP1+SE;
L6:
 CP1=OPER2;       PTB=b SEQ;
 PC=OPER2!RS2,SEQ(L2);
```

(d) After Initial Cross Block Scheduling

```
 ...                      SE=4;
 RS2=r[3];                SEQ=PC+1;     OPER2=CP1+SE;
L2:
 LV=R[CP1];
 OPER1=CP2<LV;            PTB=b TARG;
 PC=OPER1:0,TARG(L6);
 CP2=LV;
L6:
 CP1=OPER2;               PTB=b SEQ;
 PC=OPER2!RS2,SEQ(L2);    OPER2=CP1+SE;
```

(e) After Further Cross Block Scheduling

```
 ...                      SE=4;
 RS2=r[3];                SEQ=PC+1;     LV=R[CP1];     OPER2=CP1+SE;
L2
 OPER1=CP2<LV;            PTB=b TARG;
 PC=OPER1:0,TARG(L6);
 CP2=LV;
L6
 CP1=OPER2;               PTB=b SEQ;
 PC=OPER2!RS2,SEQ(L2);    LV=R[CP1];    OPER2=CP1+SE;
```

(f) After Aggressive Cross Block Scheduling

Figure 9: Example of Scheduling Effects across Basic Blocks

Figure 9(c) contains the effects after scheduling within basic blocks. There are three instructions in the first loop block, one in the second loop block, and two in the third loop block. All of the effects in the first and third loop blocks are constrained by true dependences except for the PTB assignments that have to placed in the instruction preceding the ToC. Figure 9(d) presents the effects after initial cross block scheduling. Effect (6) is moved to both of its predecessor blocks and due to lack of conflicts is placed in the first instruction within each of these basic blocks. After moving effect (6) out of the third loop block, effect (7) is then moved up one instruction within the third loop block. Figure 9(e) displays the effects after the next step in cross block scheduling. Effect (6) in the second loop block is removed as its effect is redundant since CP1 and SE is not updated between the two sets of OPER2 in Figure 9(d). Effect (6) in the first loop block is also moved into both the loop preheader and the third loop block, which is referred to as a *loop tail* since it has a transition back to the loop header. Effect (6) is placed in the same instruction as effect (9) due to the antidependence effect (6) has with effect (9).

Figure 9(f) displays the effects after aggressively applying cross block scheduling, which aggressively moves effects in two ways. If a predecessor block is a loop preheader and the current block is the header for the same loop, then the compiler allows effects to be moved into instructions to the preheader block. First, the compiler moves an effect from the loop header to both the preheader and in available slots of the tail(s) of the loop even when the effect requires a new instruction at the end of the preheader as long as the number of instructions in the loop header is decreased. Creating multiple new instructions in a loop preheader can lead to a small increase in code size. Second, load and multi-cycle effects are allowed to be moved to both the loop preheader and the loop tail from the header as long as a terminating exception can be guaranteed not to occur due to the transformation. The effect moved to the loop tail will be executed one extra time on the last loop iteration, but such a transformation is a good tradeoff when the number of instructions in the loop header is decreased. In the example effect (1) is moved into both the preheader and the loop tail (third loop block). Effect (1) is placed in the same instruction as the branch since it has a true dependence with effect (7). Our effect scheduler safely ensures that a load with a constant stride of at most eight bytes will not cause a terminating exception when the memory reference is to the run-time stack or to a global data area by ensuring that the application will not reference a new page that causes an access violation. In the example the scheduler determines that effect (1) has a constant stride of four bytes. The loop header is decreased by one instruction by performing this transformation. Note this transformation would have still been applied even if effect (1) had to be added as a separate instruction in the preheader since the number of instructions in the loop header was decreased.

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 have ported the VPO compiler [5] to the SP processor. We selected the VPO compiler 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. The C front end used is LCC [6] and is combined with a *middleware* process that converts LCC's output format into the RTL format used by VPO. We use the 17 benchmarks shown in Table 1 from the *MiBench* benchmark suite [7], 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 [8]. 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 in our simulator. Over 90% of the instructions executed are SP instructions.

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 a simple bimodal branch predictor with 256 two-bit saturating counters and a 256-entry branch target buffer (BTB). The BTB is only used for MIPS code as it is not needed for the SP. More information on the simulation infrastructure can be found in our previous work [1].

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, crc, fft, gsm

When scheduling effects, the compiler can use one of three types of template sets. *32-Templates* indicates that the compiler restricts scheduling effects to use the 32 templates that were produced by the *template selector* shown in Figure 6. For each of the 17 applications, we selected templates based on a profile generated from the compilation of the other 16 applications so that the profile used for the compilation of each application would not be affected by the application itself. *Pool-of-Templates* means the compiler could use any possible template that fits in the instruction formats shown in Figure 4 as well as complying with the structural limitations (e.g., at most one memory operation per instruction). Note that only 32 templates can actually be used due to the 5-bit ID field used to indicate the combination of effects for each template. *No-Templates* specifies that the full 81-bit instruction format was used to allow any combination of effects that abides by the SP structural limitations.

4.2 Results

Each of the graphs in this section represent the ratio between SP and MIPS. 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. The ratios rather than the raw numbers are averaged 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* simulations), we averaged the figures for all of its simulations and then used that figure for the benchmark to avoid weighing benchmarks with multiple executions more heavily.

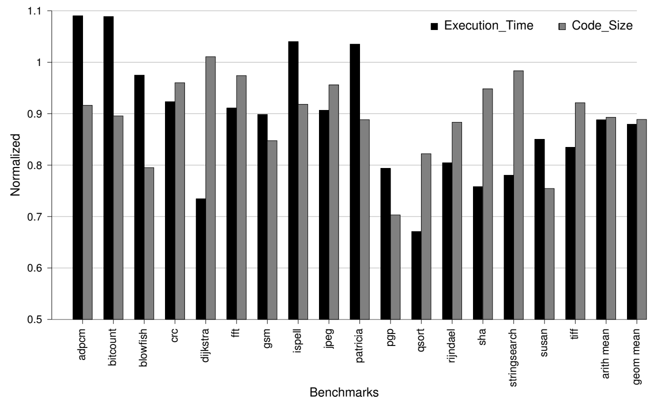

Figure 10: Normalized Execution Time and Code Size Using 32-Templates

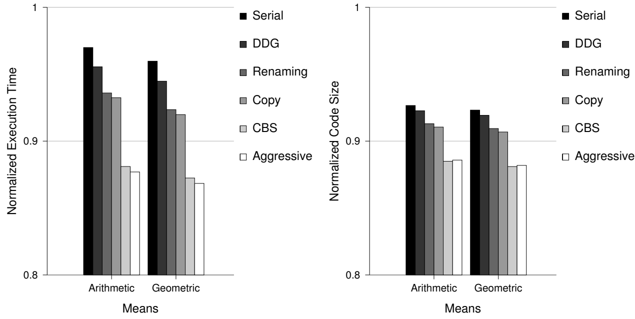

Figure 11: Normalized Execution Time Using Pool-of-Templates

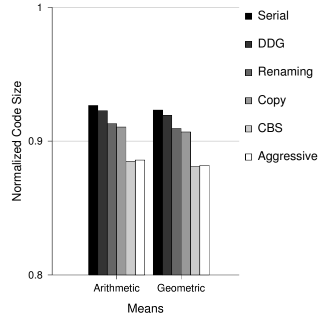

Figure 12: Normalized Code Size Using Pool-of-Templates

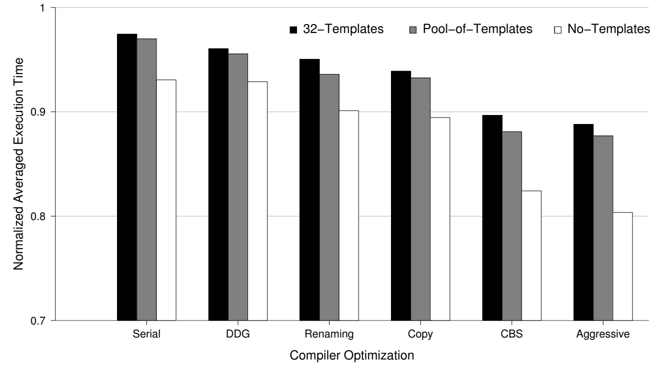

Figure 13: Performance Associated with Using Different Template Restrictions

Figure 10 contrasts execution time and code size, as compared to the MIPS baseline. For this graph we used *32-Templates* for scheduling effects. On average there was an 11.2% decline in execution cycles and a 10.7% decrease in code size. The execution cycles had a larger variance than the code size as small differences in the kernel loops of an application can cause a significant impact on performance. Only four benchmarks had execution time increases and only one benchmark had a code size increase. On average, the execution time decline was slightly larger than the code size decrease, which is not surprising given that there were several SP optimizations applied to innermost loops.

Figures 11 and 12 illustrate the impact of the different scheduling options. *Serial* represents scheduling effects in the order of the original unscheduled instructions within each basic block, which was the technique used in previous SP work [9, 10, 1]. *DDG* indicates selecting effects from a ready set based on the DDG (see Figure 7(b)-(d)). All of the remaining options use *DDG* and the other options specified after *DDG* up to and including the current option. *Renaming* means that renaming of live ranges of `RS1` to `RS2` or `OPER2` to `OPER1` when set for integer additions was also performed (see Figure 7(e)). *Copy* denotes that renaming by copying a value to a copy register was also applied (see Figure 8). *CBS* implies that cross block scheduling of effects was utilized (see Figures 9(d) and (e)). *Aggressive* signifies that aggressive techniques were used during cross block scheduling (see Figures 9(f)).

In order to isolate the impact of a scheduling option on compiled code from the impact of the selected SP templates, the results depicted in Figures 11 and 12 were produced by compiling the benchmarks with the *Pool-of-Templates* as described in Section 4.1. Evaluating a single benchmark using *32-Templates* may introduce one of two problems. First, if the template set is selected with profile data from a single scheduling option, then the template set is slightly biased towards that option. Second, if a different template set is selected for each scheduling option, then the change seen between options cannot be attributed solely to the scheduler. By using the Pool-of-Templates, we both avoid bias and highlight the contributions of the individual scheduling optimizations, while maintaining the constraint that all SP instructions must fit within a 32-bit template format.

Each scheduling option provided some benefit on average. Figure 11 shows how execution cycles are affected, as compared to the MIPS baseline. Selecting instructions from a ready set from the *DDG*, conventional *renaming* of internal registers, and performing cross block scheduling (*CBS*) provided the most benefit over *serial* scheduling. In particular, *CBS* provided the most benefit, which shows the importance of scheduling SP effects across control-flow operations. The use of *copy* renaming did not provide much benefit for two reasons. First, the compiler did not apply this renaming unless a copy register was already used in the function as saving and restoring of registers have already been applied before scheduling of effects. Second, when the compiler did already use copy registers, they often were already live at the point where copy renaming would have been useful. Determining the best number of copy registers to have in the SP architecture and whether or not all copy registers should be callee save should be investigated in the future. *Aggressive* application of cross block scheduling also did not provide much benefit. More extensive analysis to determine when an operation is safe to speculatively execute in a predecessor block may provide additional benefits. However, performing this analysis at the level of SP instructions is challenging.

Figure 12 shows the impact on code size, as compared to the MIPS baseline. To isolate the impact on code size from the impact of selecting 32 templates the presented results are generated the same way as for the execution time using the *Pool-of-Templates*. All of the options provided benefits, with the exception of *aggressive* cross block scheduling, which was expected as these techniques introduce additional instructions to move effects into loop preheaders.

Figure 13 shows how well our template selection process performs. The *32-Templates*, *Pool-of-Templates*, and *No-Templates* legends indicate the different sets of templates that can be used during effect scheduling as described in Section 4.1. *32-Templates*, which was also used for Figure 10, is the most restrictive and hence provided the least benefit. *Pool-of-Templates*, which was used to generate Figures 11 and 12, provides more flexibility as any possible combinations of effects that fits within the formats shown in Figure 4 can be used for an instruction. *No-Templates* is the most flexible as only SP structural limitations apply. The results show that our template selection process chooses 32 templates that provides close to the benefit of using the entire pool of possible templates. However, there is a significant improvement in performance when not using any templates. Thus, the use of a wider instruction format can provide significant performance benefits and we describe how such benefits may be obtained in Section 6.

4.3 Hardware Implementation

We also created an RTL implementation of the SP in order to accurately model its area, performance, and power usage. This model was compared to an RTL implementation of a traditional 5-stage pipeline capable of executing the OpenRISC instruction set. Both models are capable of executing the integer-only subset of their respective architectures, and use identical caches, adders, multipliers, and dividers. These models were synthesized to gates for a low-power 65 nm technology by ST Microelectronics.

The area for each synthesized pipeline is shown in Table 2. The SP has a smaller area primarily due to the lack of a BTB. Additionally, the SP's control logic is slightly smaller, due to the simplicity of the pipeline and reduced instruction decoding logic. On the other hand, the datapath area is significantly larger due to the large number of multiplexers required to shuttle values between registers and functional units. The SP also has an additional 32-bit adder not found in the OpenRISC, which accounts for the additional functional unit area.

Table 2: Pipeline Area Results

Component	OpenRISC	Static Pipeline
L1-IC	459406	459427
L1-DC	497640	496119
Pipeline	102828	52277
Control	4864	2970
Datapath	13625	17408
Functional Units	12411	13117
BPB	1152	1152
BTB	53117	—
Register File	17651	17626
Miscellaneous	8	4
Miscellaneous	3544	3197
Total Area	1063420	1011022

After synthesis, the netlists were simulated using a subset of the *MiBench* suite to gather switching statistics for the design. This information was fed into Synopsys Design Compiler to calculate power consumption for these benchmarks. Since support for floating-point operations adds significant complexity to processors, they were not used in this study. Thus, the benchmarks in the *MiBench* suite that use floating point were not included. The simulator used in this study does not support executing both SP instructions and MIPS instructions, since it realistically simulates the actual hardware. However, this was not a problem as the benchmarks have been modified to remove all uses of system calls.

Figure 14 shows the execution times in cycles for both the SP and the OpenRISC on the integer subset of the *MiBench* suite. On average, the SP performs nearly 7% better than the OpenRISC pipeline. Most of the improvement comes from the change in IPC, from 0.77 for the OpenRISC, to 0.95 for the SP. The SP is able to achieve this higher IPC for several reasons. First, the SP does not need to stall due to load data hazards. Second, the SP has fewer branch mispredictions because it can always provide the correct branch target without using a BTB. Third, when a branch is mispredicted, the penalty is lower due to a shorter pipeline. Additionally, the SP requires 1% fewer instructions to complete the same work, as shown in Figure 14.

Figure 15 shows the power used by each processor. The power for each processor is broken down into the following components: *Pipeline* (control and datapath), *L1 instruction cache* (SRAMs and controller), *L1 data cache* (SRAMs and controller), and *Miscellaneous* (cache arbiter, etc.). Most

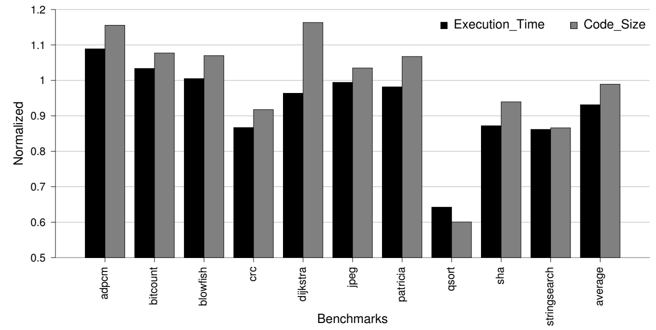

Figure 14: Normalized Execution Time and Code Size

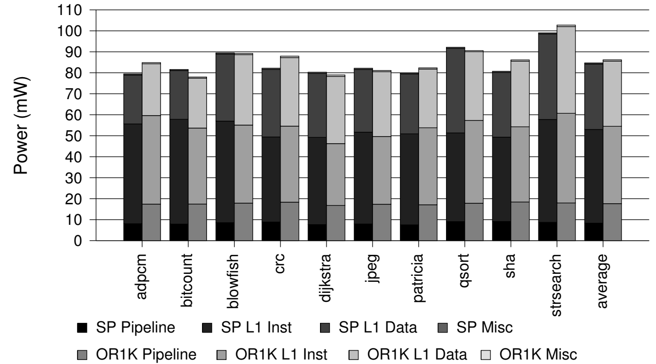

Figure 15: Processor Power

significantly, the L1 instruction cache power usage is much higher for the SP. This is explained by the higher IPC found in the SP. The increase in instruction cache power does not completely undo the power saved by the SP, which shows an average total core power reduction of nearly 2%.

Figure 16 breaks down the power used by each subcomponent of the pipelines, excluding caches. The pipeline power for the SP is broken down into the following subcomponents: *SP Register File* (register file), *SP ALU* (adders, multiplier, divider, and shifter required to implement the FU1 and FU2 effects), *SP Datapath* (datapath, including internal registers), *SP Control* (control, including instruction expander), and *SP BPB* (branch prediction buffer). The pipeline power for the OpenRISC pipeline is broken down into the following subcomponents: *OR1K Register File* (register file), *OR1K ALU* (adders, multiplier, divider, and shifter), *OR1K Datapath* (datapath, including internal registers), *OR1K Con-*

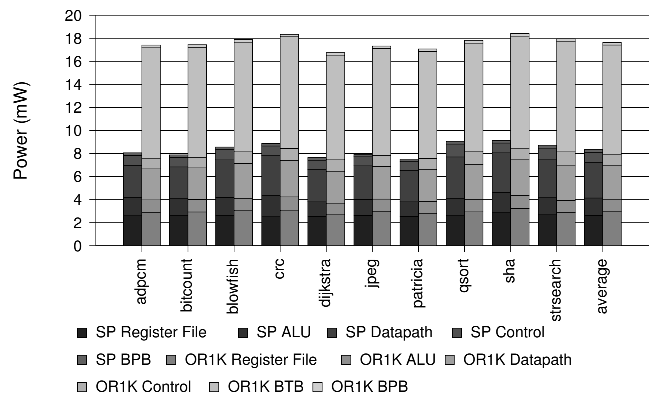

Figure 16: Pipeline-Only Power

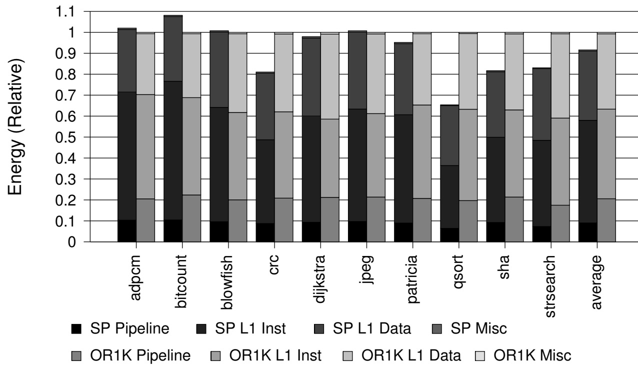

Figure 17: Processor Energy

trol (control, including instruction decoder), *OR1K BTB* (branch target buffer), and *OR1K BPB* (branch prediction buffer). The SP pipeline uses less than half the power of the OpenRISC pipeline. The vast majority of the SP's power reduction comes from the removal of the BTB, which uses over half the power of the RISC pipeline. Additional small power reductions come from reductions in register file activity. However, the SP's datapath power is somewhat higher, which is primarily due to the number of large multiplexers required. The ALU power of the SP is also higher, due to the second adder. The OpenRISC shares a single adder for both ALU operations and load/store address calculations.

Though power is slightly reduced, the performance improvement is more significant, and thus the primary benefit of the SP is actually performance, and not power reduction. However, improving performance while using the same power translates directly into energy savings. Figure 17 shows the energy used by each processor. The slight decrease in power usage, combined with the shorter execution time results in an overall reduction in energy usage of 8.5%.

5. RELATED WORK

SP instructions are most similar to horizontal micro instructions [11], however, there are significant differences. Firstly, the effects in SP instructions specify how to pipeline conventional 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. Each micro-routine can be viewed as a conventional instruction that behaves exactly the same regardless of what operations the preceding and next instructions/micro-routines perform. The exposed datapath of SP enables optimizations across consecutive instructions/micro-operations that eliminates many unnecessary operations that otherwise are performed. It has been proposed to break floating-point operations into micro-operations and optimize the resulting code [12]. However, this approach can result in a significant increase in code size. Static pipelining also bears some resemblance to VLIW [13] 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 conventional RISC instruction.

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 microarchitecture 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 [15, 16] also exposes datapath elements at the architectural level for the compiler to control. The design features a flexible datapath with an instruction decoder that is reconfigured dynamically at runtime. The transport-triggered architectures (TTAs) [17] 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, NISC, and FlexCore. 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 intermix of 16-bit and 32-bit instructions and the use of internal register queues, which increase the hardware complexity [18].

A prepare-to-branch (PTB) instruction has been previously proposed, but the use of this feature has previously required an entire instruction and thus may impact code size and performance [19]. In contrast, our PTB field only requires four bits as the target address calculation is decoupled from both the PTB field and the point of the transfer of control.

6. FUTURE WORK

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 [20], 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. This would mean that the instructions would have additional different types of effects, which means additional code size tradeoffs.

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 include using a low-level representation that violates 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 as described in this paper.

We show in this paper that SP effects can be scheduled in conventional sized 32-bit instructions to produce reasonable improvements in performance, code size, and energy usage, as compared to a conventional MIPS baseline. We demonstrate that the process of selecting templates for legal combinations of SP effects can be automated and we illustrate that a number of scheduling techniques can be applied to overcome the challenges associated with an SP architecture. With effective effect scheduling, SP architectures may be a reasonable design alternative for implementing energy efficient processors.

8. 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.

9. REFERENCES

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

[2] Baird, R., Gavin, P., Själander, M., Whalley, D., Uh, G.R.: Optimizing transfers of control in the static pipeline architecture. In: Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems, New York, NY, USA, ACM (2015) 1:1–1:10

[3] Aditya, S., Mahlke, S.A., Rau, B.R.: Code size minimization and retargetable assembly for custom epic and vliw instruction formats. ACM Transactions on Design Automation of Electronic Systems **5**(4) (2000) 752–773

[4] Aditya, S., Rau, B.R., Johnson, R.: Automatic design of vliw and epic instruction formats. Technical report, Hewlett-Packard, HP laboratories Palo Alto (2000)

[5] Benitez, M., Davidson, J.: A portable global optimizer and linker. ACM SIGPLAN Notices **23**(7) (1988) 329–338

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

[7] Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the International Workshop on Workload Characterization, Washington, DC, USA, IEEE Computer Society (2002) 3–14

[8] Austin, T., Larson, E., Ernst, D.: Simplescalar: An infrastructure for computer system modeling. Computer **35**(2) (2002) 59–67

[9] Finlayson, I., Uh, G., Whalley, D., Tyson, G.: Improving low power processor efficiency with static pipelining. In: Proceedings of the Workshop on Interaction between Compilers and Computer Architectures, Washington, DC, USA, IEEE Computer Society (2011) 17–24

[10] Finlayson, I., Uh, G., Whalley, D., Tyson, G.: An overview of static pipelining. IEEE Computer Architecture Letters **11**(1) (2012) 17–20

[11] Wilkes, M., Stringer, J.: Micro-programming and the design of the control circuits in an electronic digital computer. Mathematical Proceedings of the Cambridge Philosophical Society **49**(02) (1953) 230–238

[12] Dally, W.: Micro-optimization of floating-point operations. In: Proceedings of the Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, ACM (1989) 283–289

[13] Fisher, J.: Vliw machine: A multiprocessor for compiling scientific code. Computer **17**(7) (1984) 45–53

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

[15] Thuresson, M., Själander, M., Björk, M., Svensson, L., Larsson-Edefors, P., Stenström, P.: Flexcore: Utilizing exposed datapath control for efficient computing. Journal of Signal Processing Systems **57**(1) (2009) 5–19

[16] Själander, M., Larsson-Edefors, P.: FlexCore: Implementing an Exposed Datapath Processor. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation. (2013) 306–313

[17] Corporaal, H., Arnold, M.: Using transport triggered architectures for embedded processor design. Proceedings of the Integrated Computer-Aided Engineering **5**(1) (1998) 19–38

[18] He, Y., She, D., Mesman, B., Corporaal, H.: Move-pro: A low power and high code density TTA architecture. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Washington, DC, USA, IEEE Computer Society (2011) 294–301

[19] Bright, A., Fritts, J., Gschwind, M.: Decoupled fetch-execute engine with static branch prediction support. Technical report, IBM Research Report RC23261, IBM Research Division (1999)

[20] ARM Ltd.: ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition. (2011)