2-12-2011

Improving Low Power Processor Efficiency with Static Pipelining

Ian Finlayson
Florida State University

Gang-Ryung Uh
Boise State University

David Whalley
Florida State University

Gary Tyson
Florida State University

 DOI: 10.1109/INTERACT.2011.7

DOI: 10.1109/INTERACT.2011.7

Improving Low Power Processor Efficiency with Static Pipelining

Ian Finlayson†, Gang-Ryung Uh‡, David Whalley† and Gary Tyson†

†Department of Computer Science
Florida State University
{finlayso whalley, tyson}@cs.fsu.edu

‡Department of Computer Science
Boise State University
uh@cs.boisestate.edu

August 20, 2011

Abstract

A new generation of mobile applications requires reduced energy consumption without sacrificin execution performance. In this paper, we propose to respond to these conflictin demands with an innovative *statically pipelined* processor supported by an optimizing compiler. The central idea of the approach is that the control during each cycle for each portion of the processor is explicitly represented in each instruction. Thus the pipelining is in effect *statically* determined by the compiler. The benefit of this approach include simpler hardware and that it allows the compiler to perform optimizations that are not possible on traditional architectures. The initial results indicate that static pipelining can significantl reduce power consumption without adversely affecting performance.

1 Introduction

With the proliferation of embedded systems, energy consumption has become an important design constraint. As these embedded systems become more sophisticated, they also need a greater degree of performance. The task of satisfying the energy consumption and performance requirements of these embedded systems is a daunting task. One of the most widely used techniques for increasing processor performance is instruction pipelining. Instruction pipelining allows for increased clock frequency by reducing the amount of work that needs to be performed for an instruction in each clock cycle. The way pipelining is traditionally implemented, however, results in several areas of inefficien y with respect to energy consumption. These inefficiencie include unnecessary accesses to the register fil when the values will come from forwarding, checking for forwarding and hazards when they cannot possibly occur, latching values between pipeline registers that are often not used and repeatedly calculating invariant values such as branch targets.

In this paper, we introduce a technique called static pipelining which aims to provide the performance benefit of pipelining in a more energy-efficien manner. With static pipelining, the control for each portion of the processor is explicitly represented in each instruction. Instead of pipelining instructions dynamically in hardware, it is done statically by the optimizing compiler. There are several benefit to this approach. First, energy consumption is reduced by avoiding unnecessary actions found in traditional pipelines. Secondly, static pipelining gives more control to the compiler which allows for more fine grained optimizations for both performance and power. Lastly, statically pipelined processors have simpler hardware than traditional processors which should provide a lower production cost.

This paper is structured as follows: Section 2 introduces static pipelining at both the micro-architectural and architectural level. Section 3 discusses compilation issues with regards to static pipelining and gives a detailed example. Section 4 gives preliminary results. Section 5 reviews related work. Section 6 discusses future work. Lastly, Section 7 draws conclusions.

DOI: 10.1109/INTERACT.2011.7

2 Statically Pipelined Architecture

One of the most common techniques for improving processor performance is instruction pipelining. Pipelining allows for increased clock frequency by reducing the amount of work that needs to be performed for an instruction in each clock cycle. Figure 1 depicts a classical five stage pipeline. Instructions spend one cycle in each stage of the pipeline which are separated by pipeline registers.

Along with increasing 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 from stage to stage for all instructions, but is only used for branches.

Pipelining also introduces branch and data hazards. Branch hazards result from the fact that, when fetching a branch instruction, we won't know for several cycles what the next instruction will be. This results in either stalls for every branch, or the need for branch predictors and delays when branches are mis-predicted. Data hazards are the result of values being needed before a previous instruction has written them back to the register file. Data hazards result in the need for forwarding logic which leads to unnecessary register file accesses. Experiments with SimpleScalar [1] running the MiBench benchmark suite [6] indicate that 27.9% of register reads are unnecessary because the values will be replaced from forwarding. Additionally 11.1% of register writes are not needed due to their only consumers getting the values from forwarding instead. Additional inefficiencies found in traditional pipelines include repeatedly calculating branch targets when they do not change, reading registers whether or not they are used for the given type of instruction, and adding an offset to a register to form a memory address even when that offset is zero.

Given these inefficiencies in traditional pipelining, it would be desirable to develop a processor that avoided them, but does not sacrifice the performance gains associated with pipelining. In this paper, we introduce an architecture to meet this goal.

Figure 2 illustrates the basic idea of our approach. With traditional pipelining, instructions spend several cycles in the pipeline. For example, the `sub` instruction in Figure 2(b) requires one cycle for each stage and remains in the pipeline from cycles four through seven. Each instruction is fetched and decoded and information about the instruction flows through the pipeline, via the pipeline registers, to control each portion of the processor that will take a specific action during each cycle.

Figure 2(c) illustrates how a statically pipelined processor operates. Data still passes through the processor in multiple cycles. But how each portion of the processor is controlled during each cycle is explicitly represented in each instruction. Thus instructions are encoded to simultaneously perform actions normally associated with separate pipeline stages. For example, at cycle 5, all portions of the processor, are controlled by a single instruction (depicted with the shaded box) that was fetched the previous cycle. In effect the pipelining is determined statically by the compiler as opposed to dynamically by the hardware. Thus we refer to such a processor as statically pipelined.

2.1 Micro-Architecture

Figure 3 depicts one possible datapath of a statically pipelined processor. [1] The fetch portion of the processor is essentially unchanged from the conventional processor. Instructions are still fetched from the instruction cache and branches are predicted by a branch predictor.

The rest of the processor, however, is quite different. Because statically pipelined processors do not need to break instructions into multiple stages, there is no need for pipeline registers. In their place are a number of internal registers. Unlike pipeline registers, these are explicitly read and written by the instructions, and can hold their values across multiple cycles.

There are ten internal registers. The `RS1` and `RS2` (register source) registers are used to hold values read from the register file. The `LV` (load value) register is used to hold values loaded from the data cache. The `SEQ` (sequential address) register is used to hold the address of the next sequential instruction at the time it is written. This regis-

[1] In order to make the figure simpler, the multiplexer in the lower right hand corner has been used for three purposes. It supplies the value written to the data cache on a store operation, the value written to the register file and the value written to one of the copy registers. In actuality there may be three such multiplexers, allowing for different values to be used for each purpose.

DOI: 10.1109/INTERACT.2011.7

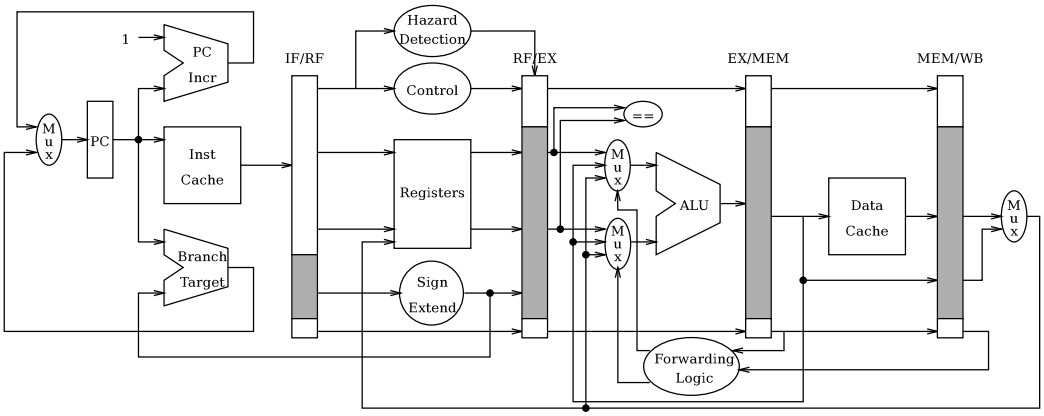

Figure 1: Simplifie Datapath of a Traditional Five Stage Pipeline

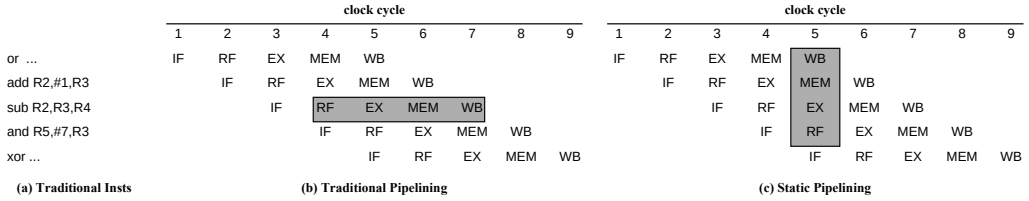

Figure 2: Traditionally Pipelined vs. Statically Pipelined Instructions

ter is used to store the target of a branch in order to avoid calculating the target. The SE (sign extend) register is used to hold a sign-extended immediate value. The ALUR (ALU result) and TARG (target address) registers are used to hold values calculated in the ALU. The FPUR (FPU result) register is used to hold results calculated in the FPU, which is used for multi-cycle operations. If the PC is used as an input to the ALU (as in a PC-relative address computation), then the result is placed in the TARG register, otherwise it is placed in the ALUR register. The CP1 and CP2 (copy) registers are used to hold values copied from one of the other internal registers. These copy registers are used to hold loop-invariant values and support simple register renaming for instruction scheduling.

Because these internal registers are part of the machine state, they must be saved and restored with the register fil upon context switches. Since these internal registers are small, and can be placed near the portion of the processor that access it, each internal register is accessible at a lower energy cost than the centralized register file Note that while the pipeline registers are read and written every cycle, the internal registers are only accessed when needed. Because these registers are exposed at the architectural level, a new level of compiler optimizations can be exploited as we will demonstrate in Section 3.

A static pipeline can be viewed as a two-stage processor with the two stages being fetch and everything after fetch. As discussed in the next sub-section, the statically pipelined instructions are already partially decoded as compared to traditional instructions. Because everything after fetch happens in parallel, the clock frequency for a static pipeline can be just as high as for a traditional pipeline. Therefore if the number of instructions executed does not increase as compared to a traditional pipeline, there will be no performance loss associated with static pipelining. Section 3 will discuss compiler optimizations for keeping the number of instructions executed as low as, or lower than, those of traditional pipelines.

Hazards due to multi-cycle operations can easily be detected without special logic to compare register numbers

DOI: 10.1109/INTERACT.2011.7

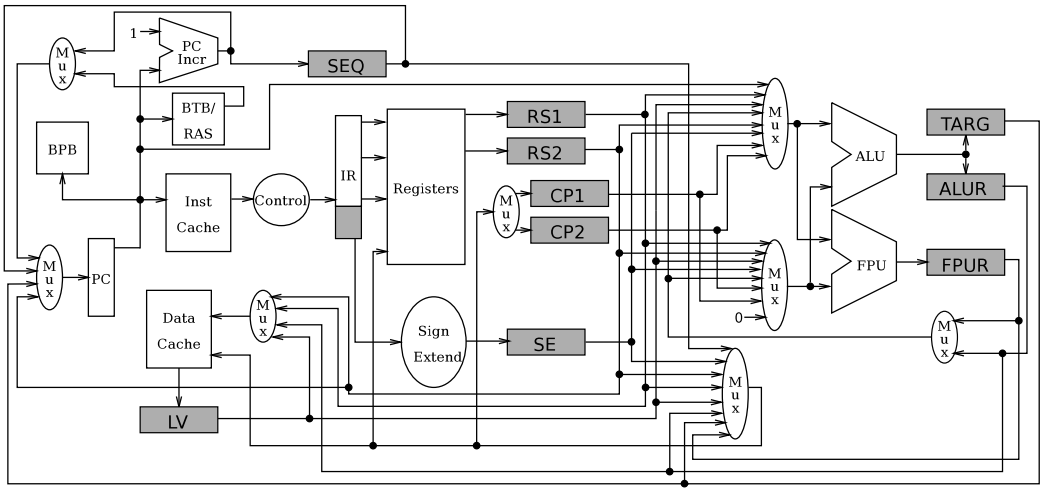

Figure 3: Possible Datapath of a Statically Pipelined Processor

from instructions obtained from pipeline registers. If during a given cycle the FPUR register is to be used as a source and the corresponding functional unit 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. Misses in the data cache can be handled in a similar fashion with the LV register.

One benefi of static pipelining is that the branch penalty is reduced to one cycle. This is because branches are resolved only one cycle after the following instruction is fetched. Interestingly, if a delay slot were employed with a static pipeline, then there would be no misprediction penalty, nor any need for branch prediction at all. In this paper, however, we do not use a delay slot and use the same fetch mechanism for the baseline MIPS and the static pipeline.

2.2 Instruction Set

The instruction set architecture for a statically pipelined architecture is quite different than one for a conventional processor. Each instruction consists of a set of effects, each of which updates some portion of the processor. The effects that are allowed in each cycle mostly correspond to what the baseline f ve-stage pipeline can do in one cycle, which include one ALU or FPU operation, one memory operation, 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 optionally be saved in the SEQ register. Lastly, the next PC can be assigned the value of one of the internal registers. If the ALU operation is a branch operation, then the next PC will only be set according to the outcome of the branch, otherwise, the branch is unconditionally taken.

In order to evaluate the architecture, we currently allow any combination of these effects to be specifie in any instruction. To specify all of these effects at one time would require 64-bit instructions, which are too wide for most low power embedded systems. In an actual implementation, only the commonly used combinations of effects would be able to be able to be specifie at a time, with a fiel in the instruction specifying which combination is used. Our preliminary analysis shows that it is practical to use 32-bit instructions with minimal loss in efficien y.

All of the effects specifie 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 fil 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.

DOI: 10.1109/INTERACT.2011.7

In a traditional architecture, when reading a value from the register file it is clear from the opcode whether that value will be used as an integer or floatin point value. This allows the instructions to "double up" on the number of available registers by having separate integer and floating-poin register files In a statically pipelined architecture, however, a register is not read in the same instruction as the arithmetic operation that uses it. Therefore to have both integer and floatin point register files we would need one extra bit for each register field To avoid this problem, we use a single register fil to hold both integer and floatin point values. Another reason for traditional architectures to use distinct register file is to simplify forwarding logic which is not an issue for this architecture. While this may increase register pressure for programs using both integer and floatin point registers, we will show in Section 3 that static pipelining reduces the number of references to the centralized register file

3 Compilation

A statically pipelined architecture exposes more details of the datapath to the compiler. This allows the compiler to perform optimizations that would not be possible on a conventional machine.

This section gives an overview of compiling for a statically pipelined architecture with a simple running example, the source code for which can be seen in Figure 4(a). The baseline we use for comparison is the MIPS architecture. The code above was compiled with the VPO [2] MIPS port, with all optimizations except instruction scheduling applied, and the main loop is shown in Figure 4(b). In this example, `r[9]` is used as a pointer to the current array element, `r[5]` is a pointer to the end of the array, and `r[6]` holds the value `m`. The requirements for each iteration of the loop are shown in Figure 4(c). [2]

We ported the VPO compiler to the statically pipelined processor. In this chapter, we will explain its function and show how this example can be compiled efficientl for a statically pipelined machine.

The process begins by firs compiling the code for the MIPS architecture with many optimizations turned on. This is done because it was found that certain optimizations, such as register allocation, were much easier to apply for the MIPS architecture than for the static pipeline. This is similar to the way in which many compilers have a platform independent and then platform dependent optimization stages.

VPO works with an intermediate representation, shown in the code listings, called "RTLs". Each generated RTL maps to one assembly language instruction on the target machine. The RTLs generated by the MIPS compiler are legal for the MIPS, but not for a static pipeline. The next step in compilation, therefore, is to break these RTLs into ones that are legal for a static pipeline.

Next, the modifie intermediate code is given as input to the compiler which produces the assembly. Figure 4(d) shows the output of the compiler run on the example above with no optimizations applied. As can be seen, the MIPS instructions are broken into the effects needed to accomplish that instruction. The dashed lines separate effects corresponding to different MIPS instructions. It's interesting to note that the instruction effects in Figure 4(d) actually correspond to what happens in a conventional pipeline, though they use field in the pipeline registers rather than internal registers. As it stands now, however, the code is much less efficien than the MIPS code, taking 15 instructions in place of 5. The next step then, is to apply traditional compiler optimizations on the initial statically pipelined code. While these optimizations have already been applied in the platform independent optimization phase, they can provide additional benefit when applied to statically pipelined instructions.

Figure 4(e) shows the result of applying copy propagation. [3] Copy propagation is an optimization which, for an assignment x = y, the compiler replaces later uses of x with y as long as intervening instructions have not changed the value of x or y. The values that were replaced by copy propagation appear in bold face in Figure 4(d).

This optimization doesn't provide any benefit on its own, but it results in assignments to registers that are never used. The next step, therefore, is to apply dead assignment elimination, the result of which can be seen in Figure 4(f). Dead assignment elimination removes as-

[2] There are f ve ALU operations because, on the MIPS, the displacement is added to the base register to form a memory address even if that displacement is 0.

[3] In actuality, VPO performs copy propagation, dead assignment elimination, redundant assignment elimination and common subexpression elimination together. They are separated here for illustrative purposes.

DOI: 10.1109/INTERACT.2011.7

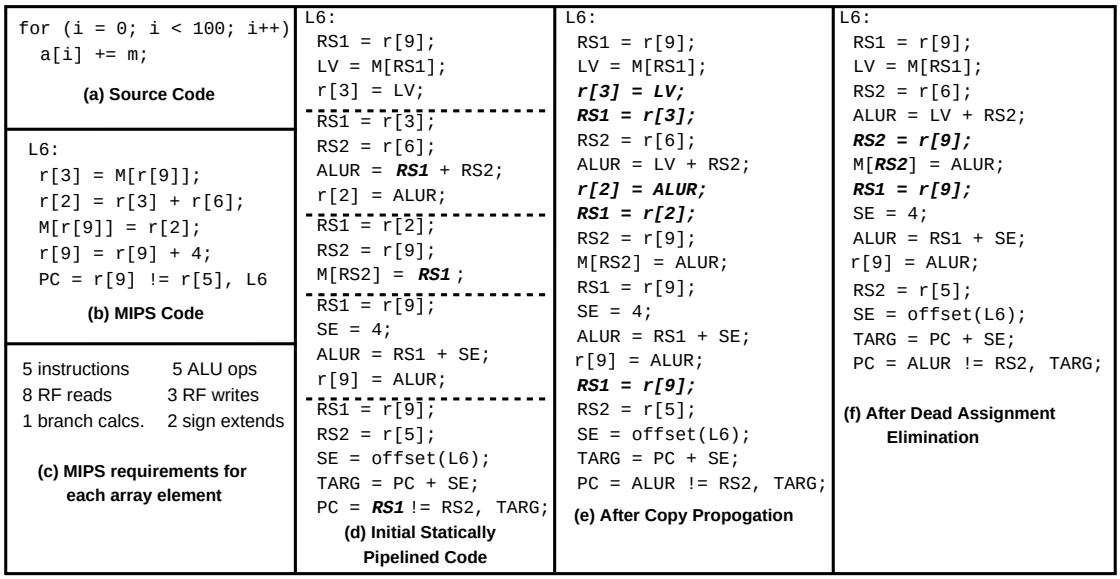

Figure 4: Example of Compiling for a Statically Pipelined Processor

signments to registers when the value is never read. The assignments that fulfil this property are shown in bold face in Figure 4(e).

The next optimization we apply is common subexpression elimination, the results of which appear in Figure 5(a). This optimization looks for instances when values are produced more than once and replaces subsequent productions of the value with the firs one. In this case, loading `r[9]` is done twice, so the compiler ruses the value in `RS1` rather than re-load the value into `RS2`. Because an internal register access is cheaper than a register fil access, the compiler will prefer the former. This is similar to the way in which compilers prefer register fil accesses to memory accesses.

We also apply redundant assignment elimination at this point. This optimization removes assignments that have been made previously so long as neither value has changed since the last assignment. In this case the assignment `RS1 = r[9];` has become redundant after dead assignment elimination, so can be removed. The RTLs affected are shown in bold face in Figure 4(f).

Because the effects that were removed have to remain in a traditional pipeline, removing them saves energy consumption over the baseline. By making these effects explicit, static pipelining gives the compiler the ability to target them. Some of these optimizations may not affect the performance after scheduling is performed, but it will affect the energy consumption. Our compiler also currently performs control fl w optimizations and strength reduction, but these did not affect the loop body in this example.

While the code generation and optimizations described so far have been implemented and are automatically performed by the compiler, the remaining optimizations discussed in this section are performed by hand, though we will automate them. The firs one we perform is loop-invariant code motion. Loop-invariant code motion is an optimization that moves instructions out of a loop when doing so does not change the program behavior. Figure 5(b) shows the result of applying this transformation. The effects that were moved outside the loop are shown in bold face in Figure 5(a). As can be seen, loop-invariant code motion also can be applied to statically pipelined code in ways that it can't for traditional architectures. We are able to move out the calculation of the branch target and also the sign extension. Traditional machines are unable to break these effects out of the instructions that utilize them so the values are needlessly calculated each itera-

DOI: 10.1109/INTERACT.2011.7

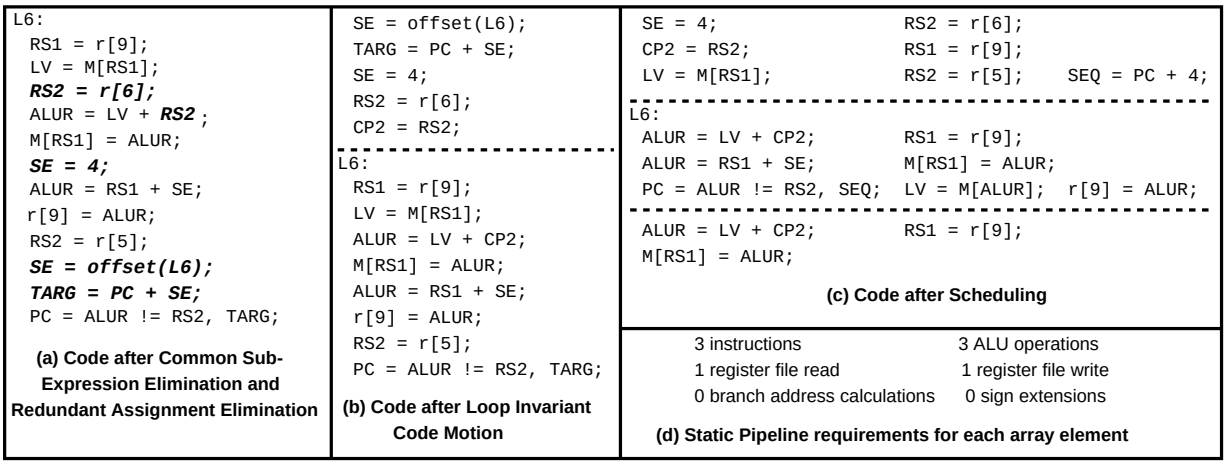

Figure 5: Example of Optimizing Code for a Statically Pipelined Processor

tion. Also, by taking advantage of the copy register we are able to move the read of `r[6]` outside the loop as well. The compiler is now able to create a more efficien loop due to its fine-graine control of the instruction effects.

While the code in Figure 5(b) is an improvement, and has fewer register fil accesses than the baseline, it still requires more instructions. This increase in execution time may offset any energy savings we achieve. In order to reduce the number of instructions in the loop, we need to schedule multiple effects together. For this example, and the benchmark used in the results section, the scheduling was done by hand.

Figure 5(c) shows the loop after scheduling. The iterations of the loop are overlapped using software pipelining [3]. With the MIPS baseline, there is no need to do software pipelining because there are no long latency operations. For a statically pipelined machine, however, it allows for a tighter main loop. We also pack together effects that can be executed in parallel, obeying data and structural dependencies. Additionally, we remove the computation of the branch target by storing it in the `SEQ` register before entering the loop.

The pipeline requirements for the statically pipelined code are shown in Figure 5(d). In the main loop, we had two fewer instructions and ALU operations than the baseline. We also had seven fewer register fil reads and two fewer register fil writes, and removed a sign extension and branch address calculation. For this example, the loop body will execute in fewer instructions and with less energy consumption.

The baseline we are comparing against was already optimized MIPS code. By allowing the compiler access to the details of the pipeline, it can remove instruction effects that cannot be removed on traditional machines. This example, while somewhat trivial, does demonstrate the ways in which a compiler for a statically pipelined architecture can improve program efficien y.

4 Evaluation

This section will present a preliminary evaluation using benchmarks compiled with our compiler and then hand-scheduled as described in the previous section. The benchmarks used are the simple vector addition example from the previous section, and the convolution benchmark from Dspstone [12]. Convolution was chosen because it is a real benchmark that has a short enough main loop to make scheduling by hand feasible.

We extended the GNU assembler to assemble statically pipe-lined instructions and implemented a simulator based on the SimpleScalar suite. In order to avoid having to compile the standard C library, we allow statically pipelined code to call functions compiled for MIPS. There is a bit in the instruction that indicates whether it

DOI: 10.1109/INTERACT.2011.7

is a MIPS or statically pipelined 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. In order to make for a fair comparison, we set the number of iterations to 100,000. For both benchmarks, when compiled for the static pipeline, over 98% of the instructions executed are statically pipelined ones, with the remaining MIPS instructions coming from calls to printf. For the MIPS baseline, the programs were compiled with the VPO MIPS port with full optimizations enabled.

Table 1 gives the results of our experiments. We report the number of instructions committed, register fil reads and writes and "internal" reads and writes. For the MIPS programs, these internal accesses are the number of accesses to the pipeline registers. Because there are four such registers, and they are read and written every cycle, this figur is simply the number of cycles multiplied by four. For the static pipeline, the internal accesses refer to the internal registers.

As can be seen, the statically pipelined versions of these programs executed significantl fewer instructions. This is done by applying traditional compiler optimizations at a lower level and by carefully scheduling the loop as discussed in Section 3. The static pipeline also accessed the register fil significantl less, because it is able to retain values in internal registers with the help of the compiler.

Instead of accessing the register file the statically pipelined code accesses the internal registers often, as shown in the table. It may appear that the only benefi of static pipelining is that the registers accessed are single registers instead part of a larger register file However, the static pipeline uses the internal registers in lieu of the pipeline registers. As can be seen in the table, the pipeline registers are accessed significantl more often than the internal registers. Additionally the pipeline registers are usually much larger than the internal registers.

While accurate energy consumption values have yet to be assessed, it should be clear that the energy reduction in these benchmarks would be significant While the results for larger benchmarks may not be quite so dramatic as these, this experiment shows that static pipelining, with appropriate compiler optimizations has the potential to be a viable technique for significantl reducing processor energy consumption.

5 Related Work

Statically pipelined instructions are most similar to horizontal micro-instructions [11], however, there are signifi cant differences. Firstly, the effects in statically pipelined instructions specify how to pipeline instructions 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 microprogrammed 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. Static pipelining also bares some resemblance to VLIW [5] 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 static pipelining approach encodes individual instruction effects that can be issued in parallel, where each effect corresponds to an action taken by a single pipeline stage of a traditional instruction.

Another architecture that exposes more details of the datapath to the compiler is the Transport-Triggered Architecture (TTA) [4]. TTAs are similar to VLIWs in that there are a large number of parallel computations specifie 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 Similar to TTAs are Coarse-Grained Reconfigurabl Architectures (CGRAs) [7]. CGRAs consist of a grid of functional units and register files Programs are mapped onto the grid by the compiler, which has a great deal of fl xibility in scheduling. Another architecture that gives the compiler direct control of the micro-architecture is the No Instruction Set Computer (NISC) [8]. Unlike other architectures, there is no fi ed ISA that bridges the compiler with the hardware. Instead, the compiler generates control signals for the datapath directly. All of these architectures rely on multiple functional units and register file to improve performance at the expense of a significan increase in code size. In contrast, static pipelining focuses on improving energy consumption without adversely affecting performance or code size.

There have also been many studies that focused on increasing the energy-efficien y of pipelines by avoiding

DOI: 10.1109/INTERACT.2011.7

Benchmark	Architecture	Instructions	Register Reads	Register Writes	Internal Reads	Internal Writes
Vector Add	MIPS	507512	1216884	303047	2034536	2034536
	Static	307584	116808	103028	1000073	500069
	reduction	**39.4%**	**90.4%**	**66.0%**	**50.8%**	**75.4%**
Convolution	MIPS	1309656	2621928	804529	5244432	5244432
	Static	708824	418880	403634	2200416	1500335
	reduction	**45.9%**	**84.0%**	**49.8%**	**58.0%**	**71.4%**

Table 1: Results of the Experimental Evaluation

unnecessary computations. One work presented many methods for reducing the energy consumption of register fil accesses [10]. One method, bypass skip, avoids reading operands from the register fil when the result would come from forwarding anyway. Another method they present 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 [9]. A strand is a sequence of instructions that has some number of inputs and only one output. The key idea here 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 which cycles its outputs back to its inputs. All of these techniques attempt to make processors running traditional instruction sets more efficient A statically pipelined processor can avoid all unnecessary register fil accesses without the need for special logic, which can negate the energy savings.

6 Future Work

The most important piece of future work is to improve the optimizing compiler. The automation of the scheduling and software-pipelining we performed by hand will allow for the evaluation of larger benchmarks. In addition we will develop and evaluate other compiler optimizations for this machine, including allocating internal registers to variables. There are also several possibilities for encoding the instructions efficientl . These options include using different formats for different sets of effects to perform, code compression and programmable decoders. Additionally, we will experiment with other architectural features such as delay slots. Another big area of future work is the development of a Verilog model. This will allow for accurate measurement of energy consumption, as well as area and timing.

7 Conclusion

In this paper, we have introduced the technique of static pipelining to improve processor efficien y. By statically specifying how instructions are broken into stages, we have simpler hardware and allow the compiler more control in producing efficien code. Statically pipelined processors provide the performance benefit of pipelining without the inefficiencie of dynamic pipelining.

We have shown how efficien code can be generated for simple benchmarks for a statically pipelined processor to target both performance and power. Preliminary experiments show that static pipelining can significantl reduce energy consumption by reducing the number of register fil accesses, while also improving performance. With the continuing expansion of high-performance mobile devices, static pipelining can be a viable technique for satisfying next-generation performance and power requirements.

Acknowledgements

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.

DOI: 10.1109/INTERACT.2011.7

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] D. Cho, R. Ayyagari, G. Uh, and Y. Paek. Preprocessing Strategy for Effective Modulo Scheduling on Multi-Issue Digital Signal Processors. In *Proceedings of the 16th International Conference on Compiler Constructions*, Braga, Portugal, 2007.

[4] H. Corporaal and M. Arnold. Using Transport Triggered Architectures for Embedded Processor Design. *Integrated Computer-Aided Engineering*, 5(1):19–38, 1998.

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

[6] 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.

[7] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurabl architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146. ACM, 2006.

[8] 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.

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

[10] J. H. Tseng and K. Asanovic. Energy-efficien 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.

[11] 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.

[12] V. Zivojnovic, J. VELARDE, and G. SCHL. C. 1994. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the Fifth International Conference on Signal Processing Applications and Technology (Oct.)*.