An Overview of 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

***Abstract*—A new generation of mobile applications requires reduced energy consumption without sacrificing execution performance. In this paper, we propose to respond to these conflicting 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 benefits 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 significantly reduce power consumption without adversely affecting performance.**

I. Introduction

With the prevalence of embedded systems, energy consumption has become an important design constraint. As these embedded systems become more sophisticated, however, they also require a greater degree of performance. One of the most widely used techniques for increasing processor performance is instruction pipelining, which 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 inefficiency with respect to energy consumption such as unnecessary register file accesses, checking for forwarding and hazards when they cannot occur, latching unused values between pipeline registers and repeatedly calculating invariant values such as branch target addresses.

In this paper, we present an overview of a technique called static pipelining [6] which aims to provide the performance benefits of pipelining in a more energy-efficient 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 benefits 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, a statically pipelined processor has simpler hardware than a traditional processor which can potentially provide a lower production cost.

This paper is structured as follows: Section 2 introduces a static pipelining architecture. Section 3 discusses compiling and optimizing statically pipelined code. Section 4 gives preliminary results. Section 5 reviews related work. Section 6 discusses future work. Lastly, Section 7 draws conclusions.

II. Statically Pipelined Architecture

Instruction pipelining is commonly used to improve processor performance, however it also introduces some inefficiencies. First is the need to latch all control signals and data values between pipeline stages, even when this information is not needed. Pipelining also introduces branch and data hazards. Branch hazards result in either stalls for every branch, or the need for branch predictors and delays when branches are mis-predicted. Data hazards result in the need for forwarding logic which leads to unnecessary register file accesses. Experiments with SimpleScalar [2] running the MiBench benchmark suite [8] 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. Because register file energy consumption is a significant portion of processor energy, these unnecessary accesses are quite wasteful [11] [9]. 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. The goal of static pipelining is to avoid such inefficiencies while not sacrificing the performance gains associated with pipelining.

Figure 1 illustrates the basic idea of our approach. With traditional pipelining, instructions spend several cycles in the pipeline. For example, the `sub` instruction in Figure 1(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 1(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 cause simultaneous actions to be performed that are 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.

Figure 2 depicts one possible datapath of a statically pipelined processor. 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 internal registers 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` registers are used to hold values read from the register file. The `LV` register is used to hold values loaded from the data cache. The `SEQ` register is used to hold the address of the next sequential instruction at the time it is written, which is used to store the target of a branch in order to avoid calculating the target address. The `SE` register is used to hold a sign-extended immediate value. The `ALUR` and `TARG` registers are used to hold values calculated in the ALU. The `FPUR` register is used to hold results calculated in the FPU, which is used for multi-

[1]Manuscript submitted: 11-Jun-2011. Manuscript accepted: 08-Aug-2011. Final manuscript received: 03-Nov-2011

Posted to the IEEE & CSDL on 11/10/2011
DOI 10.1109/L-CA.2011.26

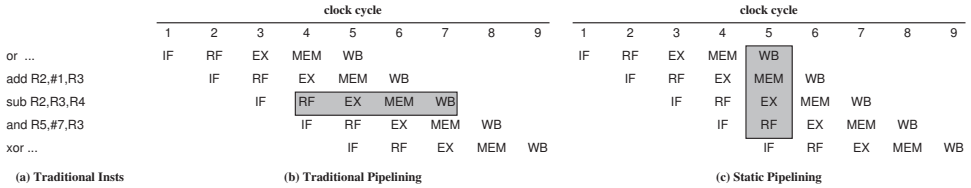

Fig. 1. Traditionally Pipelined vs. Statically Pipelined Instructions

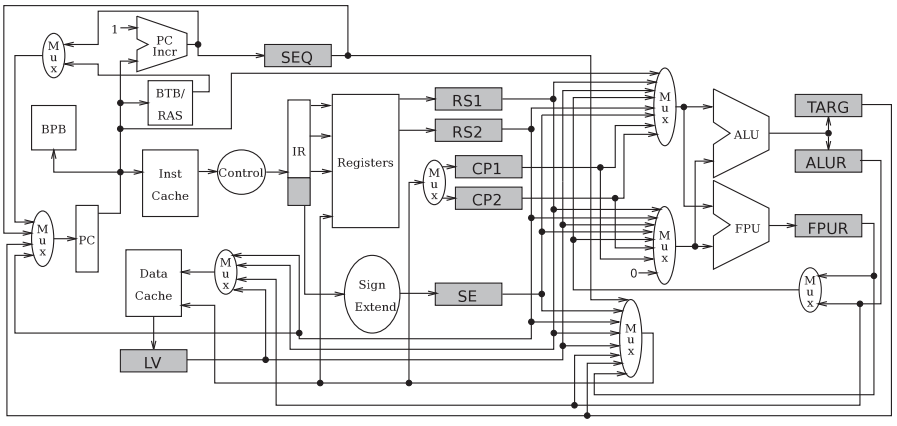

Fig. 2. Possible Datapath of a Statically Pipelined Processor

cycle operations. If the PC is used as an input to the ALU (as 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` 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. Since these internal registers are small, and can be placed near the portion of the processor that access it, they are accessible at a lower energy cost than the register file. Because more details of the datapath are exposed at the architectural level, changes to the micro-architecture are more likely to result in the need for recompilation. However this is less critical for embedded systems where the software on the system is often packaged with the hardware. 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.

Each statically pipelined 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 five-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.

To evaluate the architecture, we allow any combination of effects to be specified in any instruction, which requires 64-bit instructions. In a real implementation, only the commonly used combinations would be able to be specified at a time, with a field in the instruction specifying which combination is used. Our preliminary analysis shows that it should be practical to use 32-bit instructions with minimal loss in efficiency. The reason for this is that, while there are nine possible effects, a typical instruction will actually use far fewer. In the rare cases where too many effects are scheduled together, the compiler will attempt to move effects into surrounding instructions while obeying structural hazards and dependencies. Only when the compiler cannot do so will an additional instruction be generated for these additional instruction effects.

A static pipeline can be viewed as a two-stage processor with the two stages being fetch and everything after fetch. 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 that will attempt to keep the number of instructions executed as low as, or lower than, those of traditional pipelines.

III. Compilation

A statically pipelined architecture exposes more details of the datapath to the compiler, allowing 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 3(a). The code above was compiled with the VPO [3] MIPS port, with full optimizations applied, and the main loop is shown in Figure 3(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 3(c).

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 efficiently for a statically pipelined machine. The process begins by first 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.

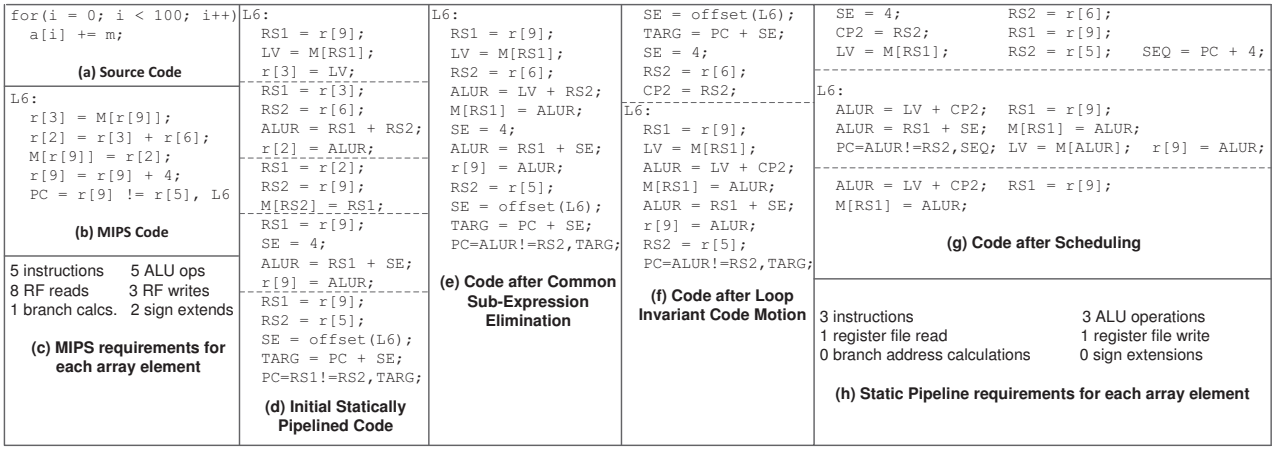

Fig. 3. Example of Compiling for a Statically Pipelined Processor

VPO works with an intermediate representation called "RTLs" where each RTL corresponds to one machine instruction on the target machine. The RTLs generated by the MIPS compiler are legal for the MIPS, but not for the statically pipelined processor. The next step in compilation, then, is to break these RTLs into ones that are legal for a static pipeline. The result of this stage can be seen in Figure 3(d). The dashed lines separate effects corresponding to the different MIPS instructions in Figure 3(b).

As it stands now, the code is much less efficient 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 benefits when applied to statically pipelined instructions. Figure 3(e) shows the result of applying common sub-expression elimination which, in VPO, includes copy propagation and dead assignment elimination. This optimization is able to avoid unnecessary instructions primarily by reusing values in internal registers, which is impossible with the pipeline registers of traditional machines. Because an internal register access is cheaper than a register file access, the compiler will prefer the former.

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 in the future. The first one we perform is loop-invariant code motion, an optimization that moves instructions out of a loop when doing so does not change the program behavior. Figure 3(f) shows the result of applying this transformation. 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 these values are repetitively calculated. Also, by taking advantage of the copy register we are able to move the read of `r[6]` outside the loop as well. We are able to create a more efficient loop due to this fine-grained control of the instruction effects.

While the code in Figure 3(f) is an improvement, and has fewer register file accesses than the baseline, it still requires more instructions. 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 3(g) shows the loop after scheduling. The iterations of the loop are overlapped using software pipelining [4]. With the MIPS baseline, there is no need to do software pipelining on this loop 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 3(h).

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 efficiency.

IV. 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 [14]. 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 pipelined 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. 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 I gives the results of our experiments. We report the number of instructions committed, register file 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 figure 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 significantly fewer instructions. This is done by applying

TABLE I
RESULTS OF THE EXPERIMENTAL EVALUATION

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%**

traditional compiler optimizations at a lower level and by carefully scheduling the loop as discussed in Section 3. The static pipeline accessed the register file significantly less, because it is able to retain values in internal registers with the help of the compiler. Also, the internal registers are accessed significantly less than the larger pipeline 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 results for larger benchmarks may not be so dramatic as these, this experiment shows that static pipelining, with appropriate compiler optimizations, has the potential to be a viable technique for significantly reducing processor energy consumption.

V. Related Work

Statically pipelined instructions are most similar to horizontal micro-instructions [13]. Statically pipelined instructions, however, specify how to pipeline instructions across multiple cycles and are fully exposed to the compiler.

Static pipelining also bears some resemblance to VLIW [7] in that the compiler determines which operations are independent. However, VLIW instructions represent multiple RISC operations to be performed in parallel, while static pipelining 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.

Other architectures that expose more details of the datapath to the compiler are the Transport-Triggered Architecture (TTA) [5], the No Instruction Set Computer (NISC) [10] and the FlexCore [12]. These architectures rely on multiple functional units and register files to improve performance at the expense of an increase in code size. In contrast, static pipelining focuses on improving energy consumption without adversely affecting performance or code size.

Another related work is the Energy Exposed Instruction Set [1] which adds some energy efficient features to a traditional architecture such as accumulator registers and tagless memory operations when the compiler can guarantee a cache hit.

VI. Future Work

One important piece of future work is to improve the optimizing compiler including the scheduling and software-pipelining. In addition we will develop and evaluate other compiler optimizations for this machine, including loop invariant code motion. Another area of future work will be encoding the instructions more efficiently. Lastly a model for estimating the energy consumption will be developed.

VII. Conclusion

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

We have shown how efficient 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 significantly reduce energy consumption by reducing the number of register file 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.

References

[1] K. Asanovic, M. Hampton, R. Krashinsky, and E. Witchel, "Energy-Exposed Instruction Sets," *Power Aware Computing*.

[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[3] M. Benitez and J. Davidson, "A Portable Global Optimizer and Linker," *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 329–338, 1988.

[4] 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.

[5] H. Corporaal and M. Arnold, "Using Transport Triggered Architectures for Embedded Processor Design," *Integrated Computer-Aided Engineering*, vol. 5, no. 1, pp. 19–38, 1998.

[6] I. Finlayson, G. Uh, D. Whalley, and G. Tyson, "Improving Low Power Processor Efficiency with Static Pipelining," in *15th Workshop on Interaction between Compilers and Computer Architectures*.

[7] J. Fisher, "VLIW Machine: A Multiprocessor for Compiling Scientific Code." *Computer*, vol. 17, no. 7, pp. 45–53, 1984.

[8] 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*. IEEE, 2002, pp. 3–14.

[9] A. Kalambur and M. Irwin, "An Extended Addressing Mode for Low Power," in *Proceedings of the 1997 international symposium on Low power electronics and design*. ACM, 1997, pp. 208–213.

[10] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 69–76.

[11] J. Scott, L. Lee, J. Arends, and B. Moyer, "Designing the Low-Power MCORE TM Architecture," in *Power Driven Microarchitecture Workshop*. Citeseer, 1998, pp. 145–150.

[12] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom, "Flexcore: Utilizing Exposed Datapath Control for Efficient Computing," *Journal of Signal Processing Systems*, vol. 57, no. 1, pp. 5–19, 2009.

[13] 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*, vol. 49, no. 02. Cambridge Univ Press, 1953, pp. 230–238.

[14] 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.)*.