Enhancing the Effectiveness of Utilizing an Instruction Register File

David Whalley, Gary Tyson
Computer Science Dept., Florida State Univ., Tallahassee, FL 32306-4530
e-mail: {whalley,tyson}@cs.fsu.edu

Abstract

This paper describes the outcomes of the NSF Grant CNS-0615085: CSR-EHS: Enhancing the Effectiveness of Utilizing an Instruction Register File. *We improved promoting instructions to reside in the IRF and adapted compiler optimizations to better utilize an IRF. We show that an IRF can decrease the execution time penalty of using an L0/filter cache while further reducing energy consumption. Finally, we introduce a tagless hit instruction cache that significantly reduces energy consumption without increasing execution time.*

1. Introduction

New processors, embedded and general purpose, often have to meet conflicting design requirements involving area, power, and performance. While a compiler optimization and/or an architectural feature may be developed to improve one design goal of a processor, it often comes at the detriment of another. An instruction register file (IRF), unlike most architectural and compiler enhancements, has been shown to simultaneously reduce power requirements, decrease code size, and improve performance [1]. This paper describes the progress and outcomes of the NSF Grant CNS-0615085, whose main goal is to enhance the effectiveness of utilizing an IRF. We also introduce a tagless hit instruction cache (IC) that reduces energy consumption without increasing execution time by guaranteeing hits in this small IC.

2. Background

Figure 1 illustrates that fields within an instruction fetched from ROM or an L1 IC can be referenced and used as indices to fetch multiple instructions from the IRF. Consider a classical five stage statically scheduled pipeline (IF→ID→EX→MEM→WB). Integration of an IRF occurs in the instruction decode (ID) stage of the pipeline. *Packed* instructions that reference the IRF specify one or more IRF indices. The IRF entries can be accessed during the first half of the ID stage, with a referenced instruction selected at the end of decode. If the instruction fetched does not specify IRF entries, then it is decoded as normal.

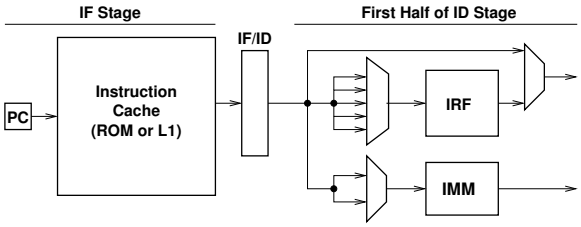

Figure 1: Accessing the IRF

We have extended the SimpleScalar PISA target [2] and the VPO (Very Portable Optimizer) compiler [3] to support packing instructions into an IRF [1]. We chose the MIPS ISA since it is commonly known and has a simple encoding. Instructions stored in memory are referred to as the Memory ISA or *MISA*. Instructions we place in the IRF are referred to as the Register ISA or *RISA*. MISA instructions that reference the RISA are designated as being *packed*.

We support two varieties of packed MISA instructions. Figure 2 shows the format for *tightly packed* instructions, allowing up to five RISA instructions to be accessed from a single MISA instruction. Our initial experiments were with a 32 entry IRF. Thus, five bits were reserved in each field of the tightly packed instructions to reference a single IRF entry. One IRF entry is reserved as a *nop* to indicate that no RISA instruction is to be executed. Hardware support halts execution of the packed instruction when a *nop* is encountered so there is no performance degradation.

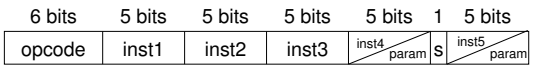

Figure 2: Tightly Packed Format

The last two 5-bit fields in the tightly packed format can also be used to refer to immediate values. We keep the 32 most frequently referenced values within an application in an immediate table, depicted in Figure 1 by IMM. This feature allows different

instructions that vary only by the value in the immediate field to be represented by the same IRF entry. There are eight different possible uses of the last two 5-bit fields (no parameterization, four different instructions referencing one parameter, three different combinations of three instructions referencing two parameters). These eight different operations are represented using four opcodes and the *S* bit.

Parameters are also used to pack branch instructions. Rather than referring to a value in the IMM table, these instructions instead treat the 5-bit parameter as a sign-extended target address displacement since the target distance may change after packing. Fortunately, most branch target distances are short. These distances are reduced even further after packing instructions into registers.

We refer to the other type of packed MISA instruction as *loosely packed* since it only references a single RISA instruction. The standard MIPS instruction R and I formats are modified to contain an index into the IRF, as shown in Figure 3. The standard instruction is first executed followed by the execution of the referenced RISA instruction. Some modifications to the MIPS instruction format were required, including reducing the size of immediate values from 16 bits to 11 bits. The vast majority of MIPS instructions in the R and I formats can be loosely packed.

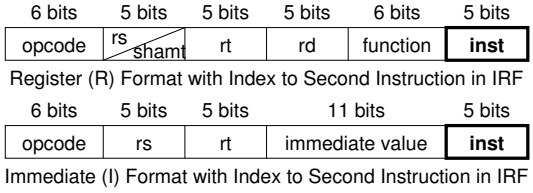

Figure 3: Loosely Packed Formats

Dynamic and static instruction profiles are used to select instructions to reduce energy consumption and decrease code size, respectively. Thus, a two pass compilation is required to reduce the dynamic number of instructions fetched from the memory system. Instructions are placed in the IRF at load time and are reloaded after each context switch.

3. Progress and Outcomes

There have been a number of enhancements that we have made to more effectively utilize an IRF. These enhancements include improving instruction promotion to the IRF, changing instruction selection and performing register re-assignment to increase instruction redundancy, and scheduling of instructions to address packing constraints [4]. We also have shown that an IRF can be used to reduce the execution penalty of an L0/filter IC while further reducing energy consumption [5].

Our initial strategy for selecting instructions to reside in the IRF was to simply choose the most frequently accessed instructions [1]. We now distinguish instructions by how many of the five RISA slots are consumed. These different classes of instructions are depicted in Figure 4. Non-packable refers to instructions that cannot support a loosely packed RISA reference, are not available via the IRF, and occupy all 5 RISA slots. Loosely packable refers to an instruction that is not available via the IRF, but has additional room for a RISA reference. The parameterized tightly packable instruction is one that is available via a combination of the IRF and parameterization. Due to referencing an IRF entry and specifying one parameter, two slots are occupied, and thus there is space for up to 3 additional RISA references. Tightly packable refers to an instruction that is available in the IRF, and does not require any parameterization. These instructions will occupy only a single slot, and thus have room for up to 4 more RISA references. With more accurate modeling we now find that is it sometimes beneficial to have multiple frequently accessed I-type instructions that differ in only their immediate value to reside in the IRF simultaneously.

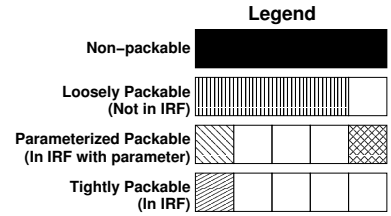

Figure 4: Different Instructions Classes

In prior IRF research, instructions were promoted to the IRF based purely on static or dynamic profile data. Dynamic profiling was used to increase IRF fetched instructions and static profiling was used to decrease code size. We now use both the dynamic and static profiles simultaneously. An embedded application developer can indicate the importance of energy consumption and code size and both profiles can be used in an attempt to meet the developer's goals.

We also enhanced instruction selection to increase the redundancy of instructions and the level of instruction packing. We now encode short jumps as branches that compare the same register for equality. Note jumps are not directly packed since they use a J format and the absolute address is unknown at compile time.

We chose equivalent parameterizable instructions over other operations. For instance, move operations were encoded as an R-type add instruction that references the hard-wired `r0`. We now encode these operations as an I-type add instruction that references an immediate value of zero, which can be a parameterizable IRF instruction. We also consistently order register operands for commutative operations.

During the second compilation pass after instructions have been promoted to the IRF, we applied a register re-assignment phase in an attempt to transform instructions to match ones that were promoted to the IRF. This optimization is accomplished by calculating the interference graph of the live ranges of registers and reassigning registers when possible and beneficial.

The final enhancement is to schedule instructions both within and across basic blocks to address packing constraints. One interesting phenomenon is that duplicating instructions across basic blocks can sometimes result in a code size decrease. Figure 5(a) shows such an example on an if-then-else code segment. Basic blocks W, X, and Y have been scheduled, and block Z is about to be scheduled. Notice that X and Y both have Z as their unconditional successor (fall-through or jump target). There are available RISA slots at the end of both basic blocks (slots a, b, c). Figure 5(b) shows instruction 1 after it has been duplicated in both predecessors of Z and is able to be combined in two separate tight packs. Block X shows that the moved instruction is actually placed before instruction 5, an unconditional jump, in order to maintain correctness. After performing intra-block scheduling on block Z, the parameterized instruction 4 is packed with instructions 2 and 3. Block Z now can be represented with a single tightly packed instruction.

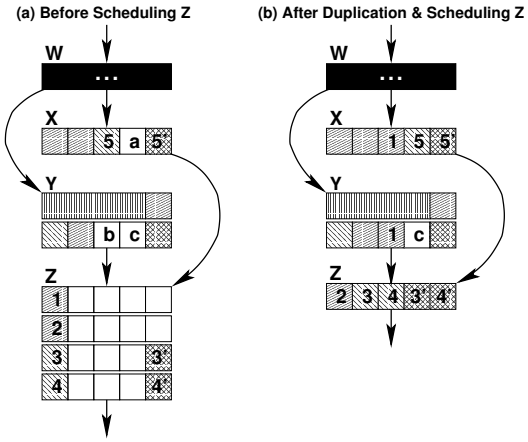

Figure 5: Duplicating Code to Reduce Code Size

Figures 6 and 7 show the results of applying these enhancements on processor energy consumption and code size, respectively. We compiled the MiBench benchmarks with the VPO compiler [3] and used SimpleScalar [2] with the Wattch extensions [6] for all results presented in this paper. The original promotion of instructions was based on a dynamic profile. Now we use a combined 50/50 dynamic and static profile. While energy consumption was not significantly altered, there is about a 6% reduction in code size. Applying the compiler optimizations significantly impacted both energy consumption and code size.

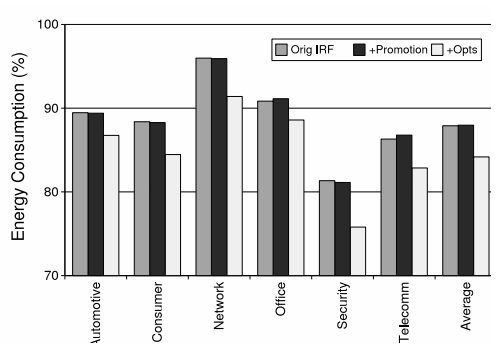

Figure 6: Enhancements on Energy Consumption

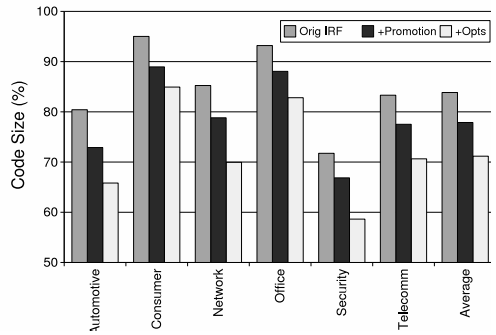

Figure 7: Enhancements on Code Size

We have also discovered that an IRF can interact effectively with other instruction fetch energy saving architectural features. For instance, we studied how an IRF interacts with an L0/filter IC [5]. Figure 8 shows pipeline diagrams for two equivalent instruction streams. In Figure 8(a), an L0 IC is being used with no IRF. The third instruction misses in the L0 cache, leading to a pipeline stall at cycle 4. Figure 8(b) shows the same L0 IC being used with an IRF. In this stream the second and third instructions are packed together into a single MISA instruction. The fourth instruction now is at the address that misses in the L0 cache. The packed instruction is fetched in cycle 2. We start the fetch of instruction 4 in cycle 3. The cache miss penalty in cycle 4 is overlapped with the

decode of the second RISA reference. The stall is avoided and the stream completes in one less cycle.

Cycle	1	2	3	4	5	6	7	8	9
Inst1	IF	ID	EX	M	WB				
Inst2		IF	ID	EX	M	WB			
Inst3			IF		ID	EX	M	WB	
Inst4					IF	ID	EX	M	WB

L0 Cache Miss at Instruction 3

Cycle	1	2	3	4	5	6	7	8	9
Inst1	IF	ID	EX	M	WB				
Pack2a		IF_{ab}	ID_a	EX	M	WB			
Pack2b				ID_b	EX	M	WB		
Inst4			IF		ID	EX	M	WB	

L0 Cache Miss at Instruction 4 with IRF

Figure 8: Overlapping Fetch with an IRF

Figure 9 shows the effect on execution time from integrating an L1 IC with an L0 IC and an IRF. An IRF decreases execution time by about 1%, mostly by reducing the working set size of applications, which decreases the number of L1 IC misses. An L0 IC increases execution time by about 16% due to the 1 cycle miss penalty for each L0 IC miss. Adding an IRF eliminates about one half of the L0 IC execution time increase. We also showed that an IRF can be used to supply additional instructions to an aggressive execution engine.

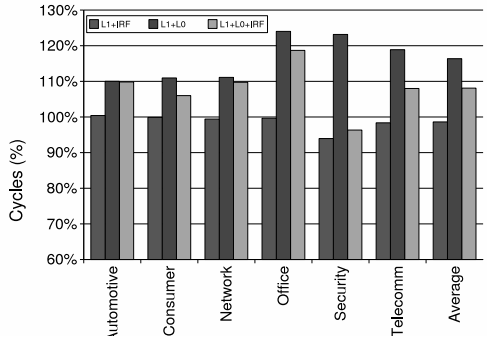

Figure 9: Execution Time with an L0 and/or IRF

Figure 10 shows the fetch energy consumption effect of integrating an L1 IC with an L1 IC and an IRF. An IRF reduces fetch energy consumption about 34% since accessing the IRF is much less expensive than accessing a much larger L1 IC. An L0 IC reduces fetch energy by about 66% since a larger percentage of instructions can be fetched from an L0 IC than an IRF. However, adding an IRF along with an L0 IC is the most effective, reducing energy consumption by about 73%.

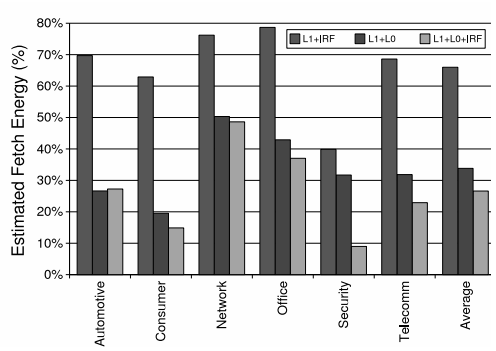

Figure 10: Fetch Energy with an L0 and/or IRF

An L0 cache can reduce energy, but can increase execution time due to its high miss rate. Any miss in the L0 IC incurs a 1-cycle miss penalty prior to fetching the appropriate line from the L1 IC. We evaluated an alternative configuration shown in Figure 11 that we call a tagless hit instruction cache (TH IC) to be used with an L1 IC [7]. Using just a few specialized metadata bits, the TH IC supplies a fetched instruction only when the instruction is *guaranteed* to reside in it. Therefore, we no longer require tag comparisons on hits, hence the term *tagless hit*. When a non-guaranteed hit is found to be in the TH IC, we refer to the access as a *false miss* and the TH IC metadata is preserved. Since the TH IC and the L1 IC are simultaneously accessed on a non-guaranteed hit, many redundant bits in the TH IC tags were able to be eliminated.

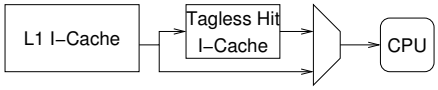

Figure 11: Tagless Hit I-Cache Configuration

In essence, the TH IC acts as a filter cache for those instructions that can be determined to be hits in the TH IC, while all instructions that cannot be guaranteed to reside in the TH IC access the L1 IC without delay. Additionally, the energy savings is greater than using a L0 IC due to the faster execution time (the TH IC has no miss penalty), the reduction in ITLB accesses (the TH IC can be accessed using bits from the portion of the virtual address that is unaffected by the translation to a physical address), as well as the elimination of tag comparisons on cache hits (since we do not need tags to verify a hit).

Figure 12 shows the hit rates of 32, 64, and 128 instruction L0 IC and TH IC configurations with four instruction lines. Note a very high percentage of TH IC hits can be guaranteed. The sum of the TH IC guaranteed hits and false misses is the same for the

equivalently sized L0 IC configurations since the TH IC really applies a different access strategy.

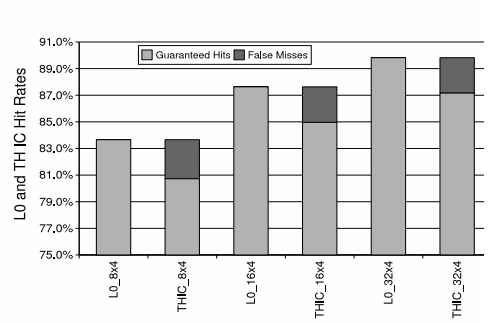

Figure 12: Hit Rates of L0 and TH IC Configurations

Figure 13 shows the processor energy consumption of the various configurations. Note that each of the TH IC configurations consume significantly less energy than its comparably sized L0 configuration. These savings are due to both more efficient access on hits and elimination of the L0 execution time penalty.

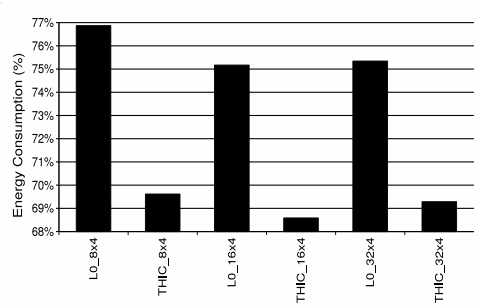

Figure 13: Energy Consumption of L0 ICs and TH ICs

4. Conclusions

In this paper we have described a number of enhancements to more effectively utilize an IRF. We applied a more accurate model and used both dynamic and static profiles to improve how instructions are selected to reside in an IRF. We adapted instruction selection and register re-assignment to increase instruction redundancy. We also perform instruction scheduling to address instruction packing constraints. These enhancements significantly reduced energy consumption and decreased code size.

We also showed that an IRF can be effectively integrated with an L0 IC. The IRF can often supply an instruction to be decoded during the extra cycle caused by an L0 IC miss. An IRF effectively cuts the execution time penalty for utilizing an L0 IC in half. Using an IRF further reduces total fetch energy since fewer instructions access the more expensive L1 IC.

Finally we introduced the TH IC organization, which serves as a filter cache for instructions that are guaranteed to be hits. This approach completely eliminates the execution time penalty incurred by a conventional L0/filter cache. Due to the high percentage of instructions that can be accessed from the TH IC, the savings in energy consumption are significant.

Lowering energy consumption and/or reducing power requirements are increasingly more important design constraints for embedded and general-purpose processors. Instruction fetch is prime area for improvement since fetches occur frequently and consume a significant portion of a processor's energy. An IRF exploits instruction redundancy and a TH IC exploits the regularity in instruction fetch behavior.

5. References

[1] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving Program Efficiency by Packing Instructions into Registers," *Proceedings of the ACM SIGARCH International Symposium on Computer Architecture*, pp. 260-271 (June 2005).

[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer* **35** pp. 59-67 (February 2002).

[3] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[4] S. Hines, D. Whalley, and G. Tyson, "Adapting Compilation Techniques to Enhance the Packing of Instructions into Registers," *ACM/IEEE Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 43-53 (October 2006).

[5] S. Hines, G. Tyson, and D. Whalley, "Addressing Instruction Fetch Bottlenecks by Using an Instruction Register File," *ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 165-174 (June 2007).

[6] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proceedings of the International Symposium on Computer Architecture*, pp. 83-94 (2000).

[7] S. Hines, D. Whalley, and G. Tyson, "Guaranteeing Hits to Improve the Efficiency of a Small Instruction Cache," *Proceedings of the ACM SIGMICRO International Symposium on Microarchitecture*, pp. 433-444 (December 2007).