Improving the Energy and Execution Efficiency of a Small Instruction Cache by Using an Instruction Register File

Stephen Hines, Gary Tyson, David Whalley
Computer Science Dept.
Florida State University
Tallahassee, FL 32306-4530
{hines,tyson,whalley}@cs.fsu.edu

Abstract

Small filter caches (L0 caches) can be used to obtain significantly reduced energy consumption for embedded systems, but this benefit comes at the cost of increased execution time due to frequent L0 cache misses. The Instruction Register File (IRF) is an architectural extension for providing improved access to frequently occurring instructions. An optimizing compiler can exploit an IRF by packing an application's instructions, resulting in decreased code size, reduced energy consumption and improved execution time primarily due to a smaller footprint in the instruction cache. The nature of the IRF also allows the execution of packed instructions to overlap with instruction fetch, thus providing a means for tolerating increased fetch latencies. This paper explores the use of an L0 cache enhanced with an IRF to provide even further reduced energy consumption with improved execution time. The results indicate that the IRF is an effective means for offsetting execution time penalties due to pipeline frontend bottlenecks. We also show that by combining an IRF and an L0 cache, we are able to achieve reductions in fetch energy that is greater than using either feature in isolation.

KEYWORDS: *Instruction Register File, Instruction Packing, Filter (L0) Instruction Caches*

1 Introduction

Recent processor design enhancements have increased demands on the instruction fetch portion of the processor pipeline. Code compression, encryption, and a variety of power-saving cache strategies can each impose performance penalties in order to obtain their benefits. These penalties are often significant, limiting the applicability of each technique to only those systems in which they are deemed critically necessary.

One important area of study, particularly for embedded systems is reducing the power and energy consumption of instruction fetch logic. This area is also becoming increasingly important for general-purpose processor design as well. It has been shown that the L1 instruction fetch logic alone can consume nearly one third of the total processor power on the StrongARM SA110 [19]. One simple technique for reducing the overall fetch power consumption is the use of a small, direct mapped filter or L0 cache [13]. The L0 cache is placed before the L1 instruction cache in such a memory hierarchy. Since the L0 cache is small and direct mapped, it can provide lower-power access to instructions at the expense of a higher miss rate. The L0 cache also imposes an extra execution penalty for accessing the L1 cache, as the L0 cache must be checked first to avoid the higher cost of accessing the L1 cache. Previous studies have shown that the fetch energy savings of a 256-byte L0 cache with 8-byte line size is approximately 68%, but the execution time is increased by approximately 46% due to miss overhead [13].

Our prior research in instruction packing can be used to diminish these performance penalties. Instruction packing is a compiler/architectural technique that seeks to improve the traditional instruction fetch mechanism by placing the frequently accessed instructions into an instruction register file (IRF) [11]. Several of these instruction registers are then able to be referenced by a single *packed* memory instruction. Such packed instructions not only reduce the code size of an application, improving spatial locality, but also allow for reduced energy consumption, since the instruction cache does not need to be accessed as frequently. The combination of reduced code size and improved fetch access can also translate into reductions in execution time.

In this paper, we explore the possibility of integrating an instruction register file into an architecture possessing a small L0 instruction cache. The nature of the IRF allows for an improved overlap between the execution and fetch of instructions, since each packed instruction essentially translates into several lower-cost fetches from the IRF. While the fetch stage of the pipeline is servicing an L0 instruction cache miss, the processor can continue fetching and executing instructions from the IRF. In this way, the IRF can potentially mask a portion of the additional latency due to a small instruction cache. Although each technique attempts to reduce overall fetch energy, we show that the approaches are orthogonal and able to be combined for improved fetch energy consumption as well as reduced performance penalties due to L0 cache misses. We believe that the IRF can be similarly applied to instruction encryption and/or code compression techniques that also affect the instruction fetch rate, in an effort to reduce the associated performance penalties.

The remainder of this paper has the following organization. First, we review the prior work on packing instructions into registers. Second, we describe how to integrate an instruction register file into a pipeline design with a small L0 instruction cache. Third, we show that combining an L0 cache with an IRF allows for even further reduced fetch energy consumption, as well as diminished execution overhead due to fetch penalties. Fourth, we examine some related work on improving the energy and execution efficiency of instruction fetch. Fifth, we outline some potential topics for future work. Finally, we present our conclusions for the paper.

2 Prior Work on Using an IRF

The work in this paper builds upon prior work on packing instructions into registers [11]. The general idea is to keep frequently accessed instructions in registers, just as frequently used data values are kept in registers by the compiler through register allocation. Placing instructions into a register file is a logical extension for exploiting two forms of locality in the instruction reference stream. It is well known that typically much of the execution time is spent in a small portion of the executable code. An IRF can contain these active regions of the program, reducing the frequency of accessing an instruction cache to fetch instructions and saving power. However, there is another type of locality that can also be exploited with an IRF. The unique number of instructions used in an application is much smaller than the possible combinations available in a 32-bit instruction set. We find that often there is a significant duplication of instructions, even for small executables. Lefurgy found that 1% of

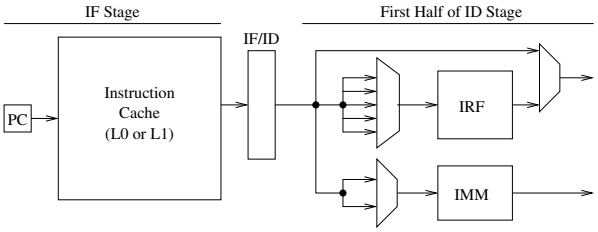

Figure 1. Decoding a Packed Instruction

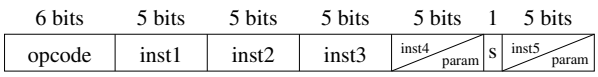

Figure 2. Packed Instruction Format

the most frequent instruction words account for 30% of program size across a variety of SPEC CINT95 benchmarks [18]. This shows that conventional instruction encoding is less efficient than it could be, which is a result of maximizing functionality of the instruction format, while retaining fixed instruction size and simple formats to ease decode. An IRF provides a second method to specify instructions, with the most common instructions having the tightest encoding. These instructions are referenced by a small index, multiples of which can easily be specified in a fixed 32-bit instruction format.

We use two new terms to help differentiate instructions in our discussion of the IRF. Instructions referenced from memory are referred to as the memory ISA or *MISA* instructions. Likewise, instructions referenced from the IRF are referred to as the register ISA or *RISA* instructions. MISA instructions that reference RISA instructions are referred to as *packed* instructions. Our ISA is based on the traditional MIPS instruction set, specifically the PISA target of SimpleScalar [1]. Figure 1 shows the use of an IRF at the start of the instruction decode stage. It is also possible to place the IRF at the end of instruction fetch or store partially decoded instructions in the IRF should the decode stage be on the critical path of the processor implementation.

Figure 2 shows the special MISA instruction format used to reference multiple RISA instructions from the IRF. These instructions are called *tightly packed* since multiple RISA instructions are referenced by a single MISA instruction. Up to five instructions from the IRF can be referenced using this format. Along with the IRF is an immediate table (IMM), as shown in Figure 1 that contains the 32 most commonly used immediate values in the program. Thus, the last two fields that could reference RISA instructions can instead be used to reference immediate values. The number of parameterized immediate values used and which RISA instructions will use them is indicated through the use of four opcodes

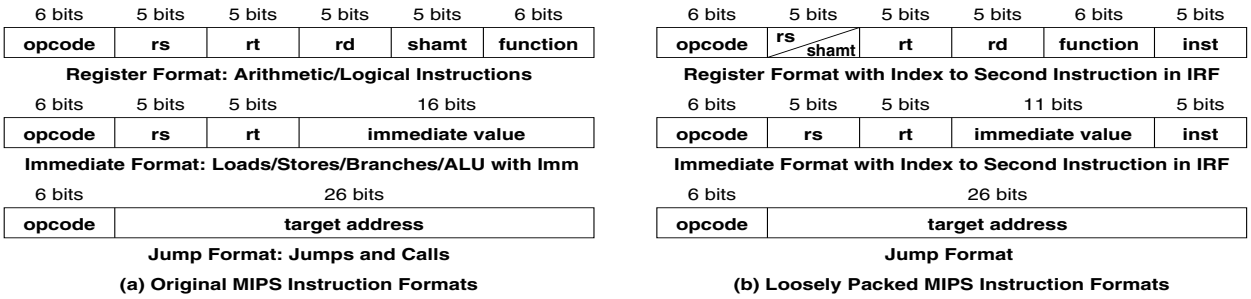

Figure 3. MIPS Instruction Format Modifications

and the 1-bit S field. The compiler uses a profiling pass to determine the most frequently referenced instructions that should be placed in the IRF. The 31 most commonly used instructions are placed in the IRF. One instruction is reserved to indicate a no-operation (*nop*) so that fewer than five RISA instructions can be packed together. Access of the RISA *nop* terminates execution of the packed MISA instruction so no performance penalty is incurred. The compiler uses a second pass to pack MISA instructions into the tightly packed format shown in Figure 2.

In addition to tightly packed instructions, the instruction set is also extended to support a *loosely packed* instruction format. Each standard MIPS instruction (with some exceptions) has 5 bits made available for an additional RISA reference. This RISA instruction is executed following the original MISA instruction. If there is no meaningful RISA instruction that can be executed, then IRF entry 0, which corresponds to a *nop*, is used. There is no performance penalty if the RISA reference is 0, since no instruction will be executed from the IRF and fetching will continue as normal. While the goal of tightly packed instructions is improved fetching of frequently executed instruction streams, the loosely packed format helps in capturing the same common instructions when they are on infrequently executed paths and not surrounded by other packable instructions. Loose packs are responsible for a significant portion of the code size reduction when profiling an application statically.

Figure 3 shows the differences between the traditional MIPS instruction formats and our loosely packed MISA extension. With R-type instructions, the *shamt* (shift amount) field can be used for a RISA reference and the various shifts can be given new function codes or opcodes. Immediate values in I-type instructions are reduced from 16 bits to 11 bits to make room for a RISA reference. The *lui* (load upper immediate) instruction is the only I-type that is adjusted differently, in that it now uses only a single register reference and the remaining 21 bits of the instruction for the upper immediate portion. This is necessary since we still want a simple method for creating 32 bit constants using the *lui* with 21 bits for an immediate and another I-type instruction containing an 11 bit immediate value. J-type instructions are modified slightly with regards to addresses in order to support a partitioning of the IRF.

For this study, we have extended the IRF to support 4 hardware windows [12], much in the same way that the SPARC data register file is organized [22]. This means that instead of using only 32 instruction registers, there are a total of 128 available physical instruction registers. Only 32 of these registers are accessible at any single point in time however, so the remaining 96 registers can be kept in a low-power mode in which they retain their values, but cannot be accessed. On a function call and/or return, the target address uses 2 bits to distinguish which instruction window we are accessing. The return address stack and all function call address pointers are updated at link-time according to which window of the IRF they will access. The IMM for each window is the same, since previous results have shown that 32 immediate values are sufficient for parameterizing most instructions that will exist in an IRF. Using two bits to specify the instruction window in an address pointer limits the effective address space available for an application, but we feel that 16 million instruction words is large enough for any reasonable embedded application.

3 Integrating an IRF with an L0 Instruction Cache

There are several intuitive ways in which an IRF and an L0 instruction cache can interact effectively. First, the overlapped fetch of packed instructions can help in alleviating the performance penalties of L0 instruction cache misses by giving the later pipeline stages meaningful work to do while servicing the miss. Second, the very nature of instruction packing focuses on the frequent access of instructions via the IRF, leading to an overall reduction in the number of instruction cache accesses. Third, the packing of instructions reduces the static code size of portions of the working set of an ap-

plication, leading to potentially fewer overall instruction cache misses.

Figure 4 shows the pipeline diagrams for two equivalent instruction streams. Both diagrams use a traditional five stage pipeline model with the following stages: IF — instruction fetch, ID — instruction decode, EX — execute, M — memory access, and WB — writeback. In Figure 4(a), an L0 instruction cache is being used with no IRF. The first two instructions (Insn1 and Insn2) execute normally with no stalls in the pipeline. The third instruction is a miss in the L0 cache, leading to the bubble at cycle 4. The fourth instruction is unable to start fetching until cycle 5, when Insn3 has finally finished fetching and made it to the decode stage of the pipeline. This entire sequence takes 9 cycles to finish executing.

Figure 4(b) shows the same L0 instruction cache being used with an IRF. In this stream, however, the second and third instructions (previously Insn2 and Insn3) are packed together into a single MISA instruction, and the fourth instruction (third MISA instruction) is now at the address that will miss in the L0 cache. We see that the packed instruction is fetched in cycle 2. The packed instruction decodes its first RISA reference (Pack2a) in cycle 3, while simultaneously we are able to start fetching instruction 4. The cache miss bubble in cycle 4 is overlapped with the decode of the second RISA instruction (Pack2b). After the cache miss is serviced, Insn4 is now ready to decode in cycle 5. In this way, sequences of instructions with IRF references can alleviate stalls due to L0 instruction cache misses. This stream finishes the same amount of work as the first stream in only 8 cycles, 1 less cycle than the version without IRF. Denser sequences of instructions (with more packed instructions) allow for even greater cache latency tolerance, and can potentially alleviate a significant portion of the latency of accessing main memory on an L1 instruction cache miss.

In previous studies, it was shown that a single 32-entry IRF can be responsible for approximately 55% of the non-library instructions fetched in an application [11]. This amounts to a significant fetch energy savings due to not having to access the L1 instruction cache as frequently. Although the L0 cache has a much lower energy cost per access than an L1 instruction cache, the IRF will still reduce the overall traffic to the entire memory hierarchy. Fewer accesses that reach into the memory hierarchy can be beneficial as energy can be conserved by not accessing the TLB or even the L0 cache tag array.

Instruction packing is inherently a code compression technique, allowing some additional benefits to be extracted from it as well. As instructions are packed together, the L0 instruction cache can potentially handle

Cycle	1	2	3	4	5	6	7	8	9
Insn1	IF	ID	EX	M	WB				
Insn2		IF	ID	EX	M	WB			
Insn3			IF		ID	EX	M	WB	
Insn4					IF	ID	EX	M	WB

(a) L0 Cache Miss at Insn3

Cycle	1	2	3	4	5	6	7	8	9
Insn1	IF	ID	EX	M	WB				
Pack2a		IF_{ab}	ID_a	EX_a	M_a	WB_a			
Pack2b				ID_b	EX_b	M_b	WB_b		
Insn4			IF		ID	EX	M	WB	

(b) L0 Cache Miss at Insn4 with IRF

Figure 4. Overlapping Fetch with an IRF

larger working sets at no additional cost. Being that the L0 cache is fairly small and direct mapped, the compressed instruction stream may be extremely beneficial in some cases, allowing for fewer L0 cache misses, which translates into performance improvements and reduced overall fetch energy consumption.

4 Experimental Evaluation

In order to evaluate the effectiveness of an IRF in eliminating the execution overhead of small caches, we focused on two main configurations. The first configuration models an embedded processor, and the second models a more advanced machine with out-of-order execution. The purpose of the second model was to see the effects that a more aggressive pipeline backend has on the small instruction cache. Our modeling environment is an extension of the SimpleScalar PISA target supporting IRF instructions [1]. Each simulator is instrumented to collect the relevant data involving instruction cache and IRF access during program execution. The SimpleScalar configuration data for each of these models is shown in Table 1.

Note that for both of these models, the L0 cache and/or the IRF/IMM are only configured if they are being evaluated. It is also important to remember that the L0 cache is not able to be bypassed in this pipeline structure, so an instruction that misses in the L0 cache but hits in the L1 cache will require 2 cycles for fetch in the embedded processor and 3 cycles in the high-end processor.

We selected a subset of the MiBench embedded benchmark suite [10] for use in each of our experiments. The MiBench suite consists of six categories, each designed

Table 1. Experimental Configurations

Parameter	Embedded	High-end
I-Fetch Queue	4 entries	8 entries
Branch Predictor	Bimodal – 128	Bimodal – 1024
Branch Penalty	3 cycles	
Fetch Width	1	2
Decode Width	1	2
Issue Style	In order	Out of order
Issue Width	1	2
Commit Width	1	2
RUU size	8 entries	16 entries
LSQ size	8 entries	
L1 D-Cache	16 KB 256 lines 16 B line 4-way assoc. 1 cycle hit	32 KB 512 lines 16 B line 4-way assoc. 2 cycle hit
L1 I-Cache	16 KB 256 lines 16 B line 4-way assoc. 1 cycle hit	32 KB 512 lines 16 B line 4-way assoc. 2 cycle hit
L0 I-Cache	256 B 32 lines 8 B line direct mapped 1 cycle hit	512 B 64 lines 8 B line direct mapped 1 cycle hit
Memory Latency	32 cycles	
Integer ALUs	1	4
Integer MUL/DIV	1	1
Memory Ports	1	2
FP ALUs	1	4
FP MUL/DIV	1	1
IRF/IMM	4 windows 32-entry IRF (128 total) 32-entry Immediate Table 1 Branch/pack	

to exhibit application characteristics representative of a typical embedded workload in that particular domain. Several of these benchmarks (Jpeg, Ghostscript, Gsm, Pgp, Adpcm) are similar benchmarks to those found in the MediaBench suite [15] used in the original evaluation of L0 caches. Table 2 shows the exact benchmarks that were used in our evaluations. For each benchmark with multiple data sets, we chose the small inputs to keep the running time of the simulations manageable.

Optimized code is generated using a modified port of the VPO compiler for the MIPS [4]. Each benchmark is profiled dynamically for instruction frequency counts and instruction packing is performed using a greedy algorithm for 4 partitions where necessary. The actual instructions available to the IRF are selected by *irfprof*,

Table 2. MiBench Benchmarks

Category	Applications
Automotive	Basicmath, Bitcount, Qsort, Susan
Consumer	Jpeg, Lame, Tiff
Network	Dijkstra, Patricia
Office	Ghostscript, Ispell, Rsynth, Stringsearch
Security	Blowfish, Pgp, Rijndael, Sha
Telecomm	Adpcm, CRC32, FFT, Gsm

our profile-driven IRF selection and layout tool. In previous studies [11], the library code was removed from the experiments to provide a fairer evaluation of the IRF mechanism. In these experiments, library code is not subject to instruction packing, however, it is modeled accurately in both the cycle time execution and fetch energy calculations. The benchmarks Basicmath, Qsort, Patricia, Stringsearch, and FFT are all dominated by library code, with more than 50% of each application executing standard C library functions. If instruction packing was performed on library code, the overall results when using the IRF should improve.

Each of the graphs in this section use the following conventions. All results are normalized to the baseline case for the particular processor model, which uses an L1 instruction cache with no L0 instruction cache or IRF. The label L1+IRF corresponds to adding an IRF but no L0 cache. The label L1+L0 corresponds to adding an L0 instruction cache, but no IRF, and finally L1+L0+IRF corresponds to the addition of an L0 instruction cache along with an IRF. Execution results are measured in the cycles returned from SimpleScalar. The fetch energy is estimated according to the following equation, for which the accuracy has been tuned using results from sim-panalyzer [20]:

$$\begin{aligned} E_{\text{fetch}} &= 10000 \times \text{Accesses}_{\text{memory}} + 100 \times \text{Accesses}_{\text{L1}} \\ &\quad + 2 \times \text{Accesses}_{\text{L0}} + \text{Accesses}_{\text{IRF}} \end{aligned}$$

These scaling factors are conservative and in line with previous work regarding instruction caches, L0 caches, and the IRF. The L0 cache requires slightly more energy to operate than the IRF due to the cache tag comparison. The IRF is only updated when the application loads, and from then on it need only be available for reading. Additionally, the inactive windows of the IRF can be kept in a lower-power state, yielding a smaller overall structure to be accessed (128 bytes).

Figure 5 shows the execution efficiency of the embedded processor. Adding an IRF to the baseline processor with no L0 cache yields only a minor performance improvement of 1.52%, mainly due to the code fitting better into

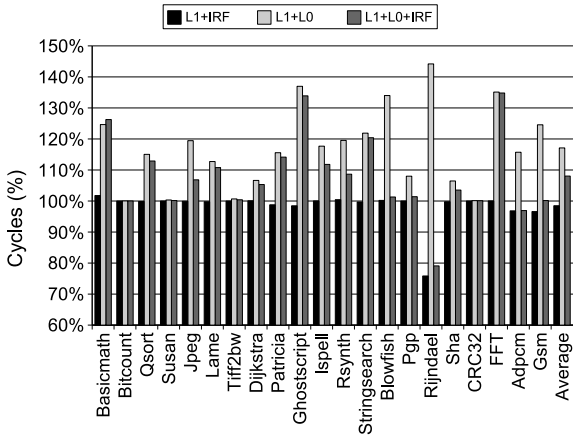

Figure 5. Embedded Execution Efficiency

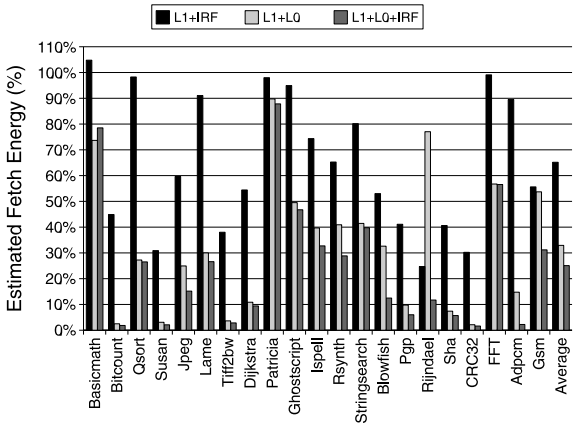

Figure 6. Embedded Fetch Energy Efficiency

the L1 instruction cache. An L0 cache degrades performance by 17.11% on average, while adding an IRF cuts this penalty to 8.04%. The overlapped fetch interaction allows the IRF to not only obtain its initial performance improvement due to a smaller footprint in the L1 instruction cache, but to also surpass it by reclaiming some of the performance penalty of the L0 instruction cache.

The fetch energy efficiency of the embedded processor is shown in Figure 6. An L1 cache with IRF yields an average fetch energy reduction of 34.83%. The L0 cache obtains reductions of 67.07%, while adding the IRF improves the energy reduction to 74.93%. The additional energy savings is a bonus, since the IRF is also used to tolerate the latency of L0 cache misses.

Conservatively assuming that fetch energy accounts for 25% of the overall processor energy, and that the non-fetch energy scales uniformly for the increased execution time, an L0 instruction cache reduces the overall energy consumption by 4%. Adding an IRF in addition to the L0 cache yields an overall energy reduction of 12.7%. If the fetch energy accounts for one third of the total processor energy, as is more likely the case for embedded systems, the overall energy savings with an L0 instruction cache is approximately 10.7%, while adding an IRF increases the savings to 19.3%.

The execution efficiency of the high-end processor is shown in Figure 7. The pipeline backend of the high-end processor is very demanding, however the L1 caches for this machine have also been modified to have a 2 cycle baseline hit time since both the fetch width and the cache sizes have been increased. An IRF can reduce the execution cycles by only 0.35%, however the performance penalty of adding an L0 cache is substantial at 35.57%. Clearly, the miss penalty of the L0 cache causes greater deterioration of performance when coupled with

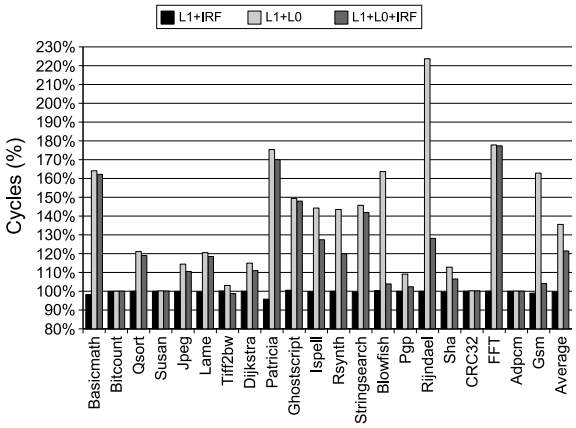

Figure 7. High-end Execution Efficiency

an aggressive pipeline backend. This phenomenon occurs since there are fewer stalls in the pipeline backend that can be overlapped with L0 instruction cache miss stalls in the pipeline frontend. The addition of an IRF and an L0 instruction cache reduces the overall performance penalty to 21.43%, showing that an even larger portion of the execution cycles can be overlapped with all associated fetch misses.

Figure 8 shows the fetch energy efficiency of the high-end processor. The addition of an IRF to the processor reduces fetch energy consumption by 33.83%. An L0 instruction cache can lower the fetch energy by 79.68% alone, or 84.57% in conjunction with an IRF. These numbers are greater than the results from the embedded system model due to the aggressive nature of this high-end processor design. In this model, the number of misfetches is greatly increased due to its highly speculative pipeline configuration, and thus the IRF and L0 can

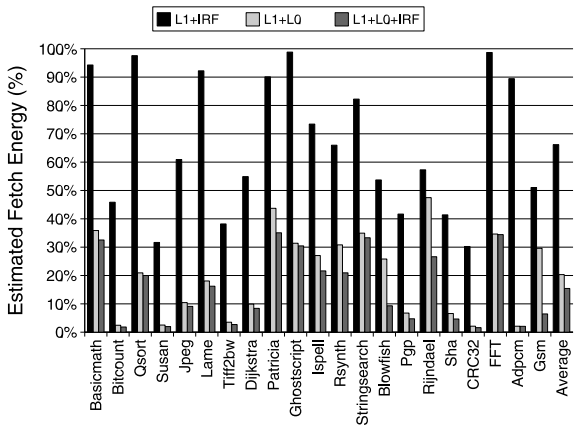

Figure 8. High-end Fetch Energy Efficiency

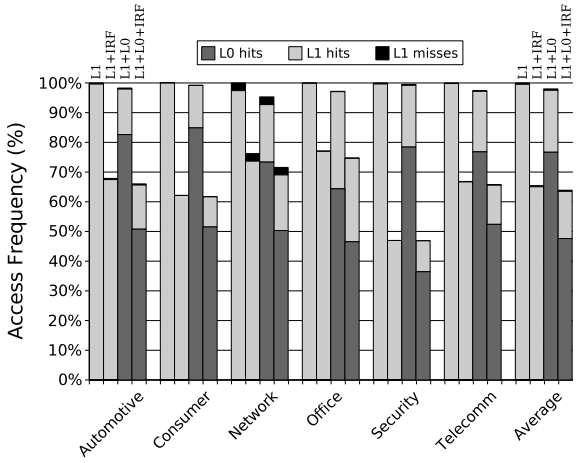

Figure 9. Embedded Cache Access Frequency

reduce the chances of having to access the L1 instruction cache and/or memory.

Figure 9 shows the frequencies at which various portions of the memory hierarchy are accessed for instructions in the embedded processor. The number of accesses are shown by category and normalized to the case where only an L1 instruction cache is in use. With just an L1 cache, these benchmarks can fetch instructions immediately from the L1 cache 99.55% of the time, yielding a very low 0.45% miss rate. In a design with an L0 cache, approximately 76.67% of the instructions fetched hit in the L0 cache, and an additional 20.87% hit in the L1 cache. The frequency of having to access main memory with an L0 cache is reduced slightly to 0.44%. The L0 instruction cache experiences a slight reduction in over-

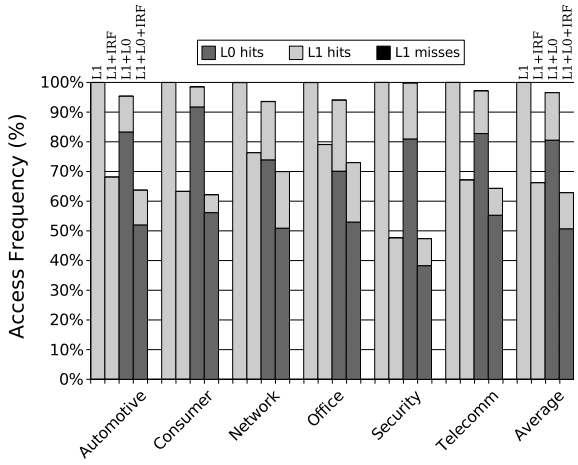

Figure 10. High-end Cache Access Frequency

all accesses due to being less aggressive on mispredictions with cache misses on some benchmarks, hence the total access frequency does not sum to exactly 100%. When adding just an IRF, the frequency of L1 cache hits is reduced to 65.05% and the frequency of misses is reduced to 0.38%, showing that greater than 34% of instructions do not need to access an instruction cache at all. Combining an L0 instruction cache with an IRF yields approximately 47.61% L0 cache hits, 15.9% L1 cache hits, and 0.37% L1 cache misses. Whereas the L0 cache converts more expensive L1 cache accesses to cheaper L0 cache accesses, the IRF is responsible for the conversion of both types of instruction cache accesses to cheaper register file accesses.

The cache access frequencies of the high-end processor are shown in Figure 10. The larger L1 instruction cache hits 99.97% of the time, missing on only 0.03% of accesses in the baseline architecture. With the use of an L0 instruction cache, there is an L0 hit 80.50% of the time, an L1 hit 16.03% of the time, and an L1 miss 0.03% of the time. The use of an IRF with just an L1 instruction cache allows for the L1 cache to hit on 66.18% of instruction fetches, and miss on only 0.02% of fetches. All other fetches are handled by the IRF. Combining the L0 instruction cache with an IRF reduces the L0 cache hits to 50.64%, the L1 cache hits to 12.20%, and the L1 cache misses to 0.02%.

Instruction packing can also reduce the static code size of an application. Figure 11 shows the normalized code size for each of the packed executables. Since we do not pack instructions in library routines, we have removed their impact on the static code size results. Overall, however we are able to reduce the actual compiled ex-

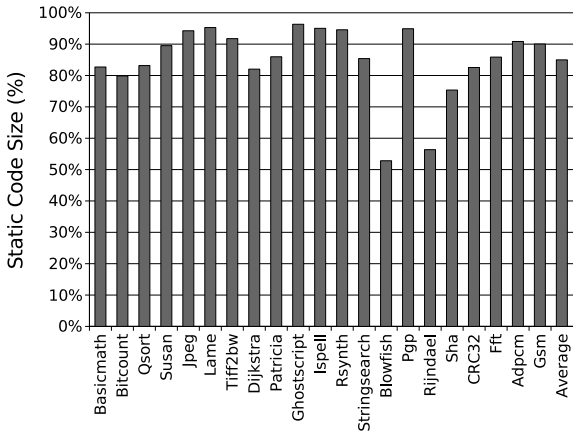

Figure 11. Reducing Static Code Size

ecutable size by 15% on average. Many of the security benchmarks experience significant reductions in static code size, since the majority of their code is dominated by similar encryption and decryption routines composed of the same fundamental instruction building blocks. These instructions can then be placed in the IRF and referenced in various combinations to construct the necessary functionality for each similar routine.

5 Related Work

Instruction and data caches are often separated for performance reasons, particularly with respect to handling the diverse behavior and request patterns for each. Another approach to reducing cache energy requirements is to further subdivide the instruction cache into categories based on execution frequency [2, 3]. Frequently executed sections of code are placed into a smaller, low-power L-cache that is similar in structure to the L0 cache discussed in this paper. The bulk of the remaining code is only accessible through the standard L1 instruction cache. Code segments for each cache are separated in the executable, and a hardware register denotes the boundary between addresses that are serviced by the L-cache and addresses that are serviced by the L1 cache. The splitting of these lookups provides a substantial reduction in the L-cache miss rate. A 512-byte L1 cache provides a 15.5% reduction in fetch energy, while also obtaining a small reduction in execution time due to improved hit rate. However, the L-cache scheme is limited in that it cannot easily scale to support longer access times from an L1 instruction cache.

Lee et al. proposed using a small cache for executing small loops with no additional transfers of control other than the loop branch [16]. Instruction are normally fetched from the L1 cache, but a short backward branch (sbb) triggers the loop cache to begin filling. If the same sbb is then taken on the next loop iteration, instructions can be fetched from the small loop cache structure instead of the L1 cache. When there is a different taken transfer of control, or the loop branch is not taken, the loop cache returns to its inactive state and resumes fetching normally from the L1 instruction cache. Since the loop cache is tagless and small (usually 8-32 instructions), the total fetch energy can be reduced by approximately 15%. The loop cache was later extended to support longer loops by adding the ability to partially store and fetch portions of a loop [17]. Another improvement to the loop cache is the use of preloading and hybridization [9]. Preloading allows the loop cache to contain the same instructions for the life of the application, while hybridization refers to a loop cache that can operate in a dynamic mode as well as preloaded. A hybrid loop cache can reduce the total instruction fetch energy by 60-70%.

An alternate method for mitigating the performance penalty of L0 caches is to provide a bypass that allows direct reading from the L1 cache in some cases. It has been shown that with a simple predictor, the L0 cache performance penalty can be dropped to 0.7% on a 4-way superscalar machine with only a small increase in fetch energy [21]. However, L0 caches are primarily used for reducing the fetch energy of embedded systems, which fetch and execute no more than one instruction per cycle.

The zero overhead loop buffer (ZOLB) is another hardware technique for reducing instruction fetch energy for small loops [7]. The main difference between a ZOLB and a loop cache is that a ZOLB is explicitly loaded using special instructions regarding the number of instructions in the loop and the number of iterations to execute. Similar to the loop cache, the ZOLB is limited in size, and can have no other transfers of control beyond the loop branch. Additionally, information regarding the number of iterations executed by the loop must be known at the time the loop is entered. Although the primary benefit of the ZOLB is fetch energy reduction, it can also provide small improvements in execution time, since loop variable increment and compare instructions are no longer necessary.

6 Future Work

There exist many areas for exploration in the design of high-performance, low-power instruction fetch mechanisms. Tolerating increased fetch latencies is one strength of the IRF, however, the instruction selection and packing algorithms have not been tuned specifically

to focus on reducing L0 cache misses. Various heuristics can be used to select IRF candidates in the areas of a program where an L0 cache miss is likely to occur. Similar heuristics can be developed to support IRF packing combined with other architectural techniques that affect fetch latency.

There are also several popular techniques that incur performance penalties due to reduced spatial locality, which may be able to be offset by the addition of an IRF. Techniques such as procedural abstraction [8, 6, 5], and echo factoring [14] seek to reduce the code size of an application by replacing common sequences of instructions with calls to extracted subroutines. However, the added function calls and returns can greatly impact the spatial locality of an application, in addition to requiring more instructions to execute normally. The IRF can be applied similarly in these cases, to reduce the impact that the cache misses and additionally executed instructions have on a compressed executable's performance. Tamper-proofed and encrypted executables experience similar performance penalties when moving code into private caches, and as such might also be able to reduce the performance impact with the addition of an IRF.

7 Conclusions

In this paper we have evaluated the interactions between a small low-power L0 instruction cache and an IRF. Our experiments focused on embedded systems for which code size, power and performance design constraints are often very stringent. We also looked at the effects of an IRF in offsetting similar performance penalties due to an L0 cache for a very aggressive pipeline backend. The use of an IRF and associated packed instructions allows a portion of the fetch miss latency of an L0 instruction cache to be tolerated. Additionally, both the L0 cache and the IRF can interact such that the fetch energy consumption is further reduced. Finally, the use of instruction packing, which is a fundamental component of the IRF microarchitecture, allows for significant reductions in the overall static code size of an application. The combination of these three improvements in standard design criteria makes an L0 cache with an IRF very attractive for use in embedded systems.

8 Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants EIA-0072043, CCR-0208892, CCR-0312493, and CCF-0444207.

References

[1] AUSTIN, T., LARSON, E., AND ERNST, D. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer 35* (February 2002), 59–67.

[2] BELLAS, N., HAJJ, I., POLYCHRONOPOULOS, C., AND STAMOULIS, G. Energy and performance improvements in a microprocessor design using a loop cache. In *Proceedings of the 1999 International Conference on Computer Design* (October 1999), pp. 378–383.

[3] BELLAS, N. E., HAJJ, I. N., AND POLYCHRONOPOULOS, C. D. Using dynamic cache management techniques to reduce energy in general purpose processors. *IEEE Transactions on Very Large Scale Integrated Systems 8*, 6 (2000), 693–708.

[4] BENITEZ, M. E., AND DAVIDSON, J. W. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation* (1988), ACM Press, pp. 329–338.

[5] COOPER, K., AND MCINTOSH, N. Enhanced code compression for embedded risc processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1999), pp. 139–149.

[6] DEBRAY, S. K., EVANS, W., MUTH, R., AND DESUTTER, B. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems 22*, 2 (March 2000), 378–415.

[7] EYRE, J., AND BIER, J. DSP processors hit the mainstream. *IEEE Computer 31*, 8 (August 1998), 51–59.

[8] FRASER, C. W., MYERS, E. W., AND WENDT, A. L. Analyzing and compressing assembly code. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction* (June 1984), pp. 117–121.

[9] GORDON-ROSS, A., COTTERELL, S., AND VAHID, F. Tiny instruction caches for low power embedded systems. *Trans. on Embedded Computing Sys. 2*, 4 (2003), 449–481.

[10] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization* (December 2001).

[11] HINES, S., GREEN, J., TYSON, G., AND WHALLEY, D. Improving program efficiency by packing instructions into registers. In *Proceedings of the 2005 ACM/IEEE International Symposium on Computer Architecture* (2005), IEEE Computer Society, pp. 260–271.

[12] HINES, S., TYSON, G., AND WHALLEY, D. Reducing instruction fetch cost by packing instructions into register windows. In *Proceedings of the 38th annual ACM/IEEE International Symposium on Microarchitecture* (2005), IEEE Computer Society.

[13] KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. H. The filter cache: An energy efficient memory structure. In *Proceedings of the 1997 International Symposium on Microarchitecture* (1997), pp. 184–193.

[14] LAU, J., SCHOENMACKERS, S., SHERWOOD, T., AND CALDER, B. Reducing code size with echo instructions. In *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (2003), ACM Press, pp. 84–94.

[15] LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 330–335.

[16] LEE, L., MOYER, B., AND ARENDS, J. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of the International Symposium on Low Power Electronics and Design* (1999), pp. 267–269.

[17] LEE, L., MOYER, B., AND ARENDS, J. Low-cost embedded program loop caching — revisited. Tech. Rep. CSE-TR-411-99, University of Michigan, 1999.

[18] LEFURGY, C. R. *Efficient execution of compressed programs*. PhD thesis, University of Michigan, 2000.

[19] MONTANARO, J., WITEK, R. T., ANNE, K., BLACK, A. J., COOPER, E. M., DOBBERPUHL, D. W., DONAHUE, P. M., ENO, J., HOEPPNER, G. W., KRUCKEMYER, D., LEE, T. H., LIN, P. C. M., MADDEN, L., MURRAY, D., PEARCE, M. H., SANTHANAM, S., SNYDER, K. J., STEPHANY, R., AND THIERAUF, S. C. A 160-mhz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Tech. J. 9*, 1 (1997), 49–62.

[20] SIMPLESCALAR-ARM POWER MODELING PROJECT. http://www.eecs.umich.edu/∼panalyzer.

[21] TANG, W., VEIDENBAUM, A. V., AND GUPTA, R. Architectural adaptation for power and performance. In *Proceedings of the 2001 International Conference on ASIC* (October 2001), pp. 530–534.

[22] WEAVER, D., AND GERMOND, T. *The SPARC Architecture Manual*, 1994.