Addressing Instruction Fetch Bottlenecks 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

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, like those experienced by encrypted ICs as well as the presence of low-power L0 caches. Although previous research has focused on the direct benefits of instruction packing, this paper explores the use of increased fetch bandwidth provided by packed instructions. Small L0 caches improve energy efficiency but can increase execution time due to frequent cache misses. We show that this penalty can be significantly reduced by overlapping the execution of packed instructions with miss stalls. The IRF can also be used to supply additional instructions to a more aggressive execution engine, effectively reducing dependence on instruction cache bandwidth. This can improve energy efficiency, in addition to providing additional flexibility for evaluating various design tradeoffs in a pipeline with asymmetric instruction bandwidth. Thus, we show that the IRF is a complementary technique, operating as a buffer tolerating fetch bottlenecks, as well as providing additional fetch bandwidth for an aggressive pipeline backend.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation; compilers; optimization; E.4 [*Coding and Information Theory*]: Data Compaction and Compression—program representation; C.1 [*Computer Systems Organization*]: Processor Architectures

General Terms Experimentation, Measurement, Performance

Keywords Instruction Register File, Instruction Packing, L0/Filter Cache

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'07 June 13–16, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-632-5/07/0006. . . $5.00

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. In order to obtain improved levels of instruction processing throughput, greater numbers and varieties of functional units are being placed on chip, resulting in increased demands on instruction fetch.

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) [17]. Several of these instruction registers are then able to be executed 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. Although the previous research has focused on many of these explicit benefits of instruction packing, the interaction of employing instruction registers with other power reduction techniques has not been fully investigated. Nor has the effect of IRF encoding been examined on more complex superscalar embedded processors with more aggressive instruction fetch requirements.

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 in 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 [29]. One simple technique for reducing the overall fetch power consumption is the use of a small, direct mapped filter or L0 cache [21]. 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 an 8-byte line size is approximately 68%, but the execution time is increased by approximately 46% due to miss overhead [21].

In this paper, we explore the possibility of integrating an instruction register file into an architecture possessing a small L0 instruction cache. It is our belief that instruction packing can be used to

diminish these performance penalties. 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 decoding 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 both an L0 instruction cache and an IRF attempt to reduce overall fetch energy, we show that these two architectural features are orthogonal and are able to be combined to further improve fetch energy consumption as well as reduce performance penalties due to L0 cache misses. We believe that the IRF can be similarly applied to instruction encryption [31] and/or code compression [26, 10] techniques that also affect the instruction fetch rate, in an effort to reduce the associated performance penalties.

We also investigate the improved fetch of instructions using an IRF in a superscalar machine with asymmetric instruction bandwidth. In these machines, the number of instructions fetched may be less than the amount of instructions that can be decoded, executed and committed in a single cycle. Although traditional instruction fetch may be unable to effectively utilize the increased execution bandwidth, the IRF allows us to often exploit the additional bandwidth when needed. This is similar in many ways to front-end throttling [28, 4, 2], which is a technique that seeks to reduce processor energy requirements by not aggressively fetching highly speculative instructions in regions with low instruction level parallelism (ILP). Whereas pipeline throttling actually limits the number of instructions fetched and decoded in a single cycle, the use of an IRF allows the fetch of instructions to remain constant, while supplying additional instructions to decode in regions of dense packing and high ILP.

This paper makes the following contributions:

- It evaluates the performance potential of employing both instruction registers and L0 or filter caches, showing that these low-power enhancements can be combined in a synergistic manner to reduce fetch energy more than previous techniques.
- It proposes new pipeline configurations that decouple fetch width from the width of the remainder of the pipeline. These new decoupled configurations give the architect greater flexibility in realizing solutions that meet conflicting performance and energy goals.

The remainder of this paper is organized as follows. 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 investigate the potential of instruction packing in reducing instruction fetch bandwidth, while still adequately supplying instructions to an aggressive pipeline backend. Fourth, we describe our experimental setup and present some results regarding the IRF and improving execution and energy efficiency. Fifth, we examine some related work on improving the energy and execution efficiency of instruction fetch. Sixth, we outline some potential topics for future research. Finally, we present our conclusions for the paper.

2. Instruction Packing with an Instruction Register File

The work in this paper builds upon prior work on packing instructions into registers [17, 18, 19]. 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

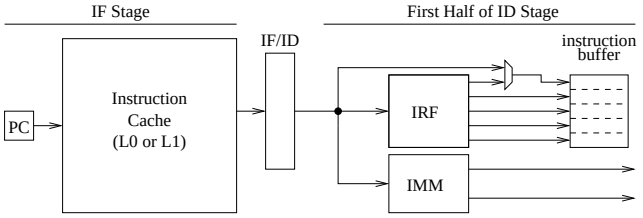

Figure 1. Decoding a Packed Instruction

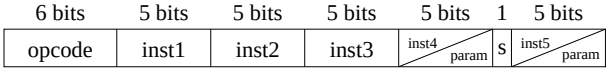

Figure 2. Packed Instruction Format

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 number of unique instructions used in an application is much smaller than the total possible combinations available with a 32-bit instruction set. Often there is a significant duplication of instructions, even for small executables. Lefurgy found that 1% of the most frequent instruction words account for 30% of program size across a variety of SPEC CINT95 benchmarks [27]. 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.

Two terms are useful in helping to differentiate instructions when discussing an architecture that supports an 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. The ISA is based on the traditional MIPS instruction set, specifically the PISA target of SimpleScalar [3]. 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 if the decode stage is 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. In Figure 1, only a single instruction is shown as being fetched from the instruction cache and fed through the IRF, however superscalar designs are also possible by adding additional read ports to the various structures shown. Along with the IRF is an immediate table (IMM), containing 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 and the 1-bit S field. The compiler uses a profiling pass to determine the most fre-

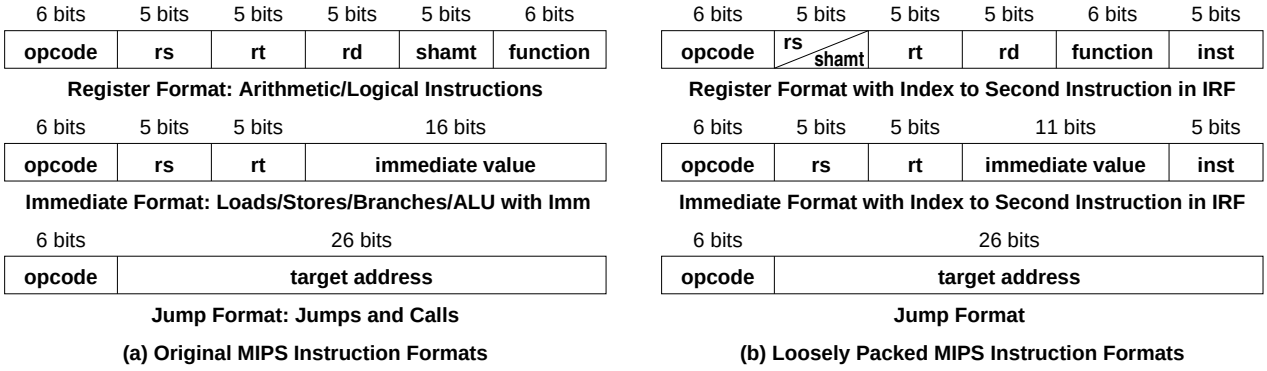

Figure 3. MIPS Instruction Format Modifications

quently 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 few 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 useful 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 the 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 partitioning of the IRF.

For this study, the IRF has been extended to support 4 hardware windows [18], much in the same way that the SPARC data register file is organized [33]. 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 function addresses 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 window in an address pointer limits the effective address space available, but we feel that 16 million instruction words is sufficiently 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 useful 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 application, 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

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

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 an L2 cache or main memory on an L1 instruction cache miss.

In previous studies, it was shown that approximately 55% of the non-library instructions fetched in an application can be accessed from a single 32-entry IRF [17]. 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 also be beneficial as energy can be conserved by not requiring access of 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 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. It is also apparent that other non-pipelinable fetch latencies can similarly be ameliorated through the use of an IRF. For example, a 2-cycle L1 instruction cache that is performing code decompression or decryption would also be able to execute additional packed instructions while stalling for the remainder of a cache access.

4. Decoupling Instruction Fetch in a Multi-issue Pipeline

Modern superscalar processor designs often provide similar instruction bandwidth throughout the pipeline. In the ideal case, instructions should flow readily from one stage to the next. Often the number and types of functional units available in a given processor are designed to handle pipeline stalls due to long latency operations. Load-store queues and additional ALUs can allow further instructions to continue flowing through the pipeline without being stalled. Although computer architects can vary the number and size of these functional units in a given pipeline, the maximum number of instructions that can be fetched, decoded/dispatched, issued, and committed are often the same.

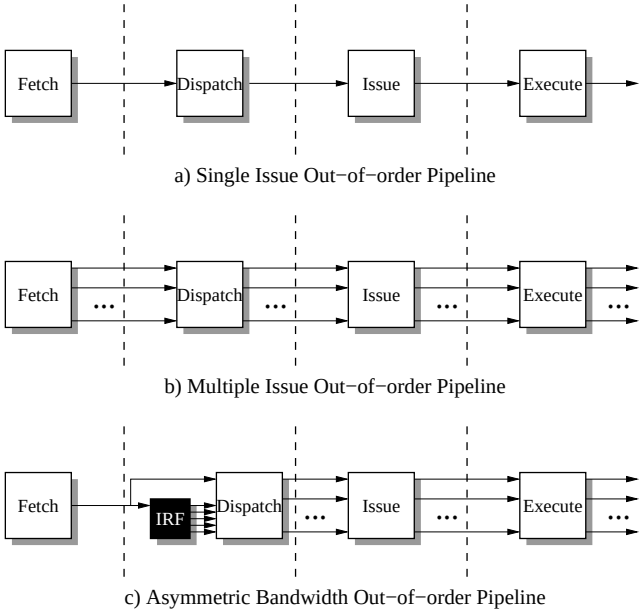

Figure 5. Decoupling Instruction Fetch in an Out-of-Order Pipeline

This makes sense when considering steady state flow of instructions through the pipeline, but can become problematic when accounting for the effects of instruction cache misses, branch misprediction recovery, alignment issues, etc. There are points where instruction fetch becomes the performance bottleneck for short periods of time, and in aggregate, these can impact the overall processor performance. Constraining the fetch width to match the rest of the pipeline imposes an artificial limitation on the design space exploration for a new architecture, particularly for embedded processors where die area, power consumption, execution time and packaging costs may be highly restrictive.

Figure 5 shows the first few pipeline stages for several out-of-order configurations. Note that each stage can be further subdivided as necessary (e.g. different cycle times for different functional units). The simplest single issue out-of-order pipeline is shown in Figure 5a. Figure 5b shows a multiple issue pipeline. As more functional units become available on chip, heavy demands can be placed issue, dispatch and fetch. Techniques already exist for appropriately scaling the issue queue [13], however the increasing pressure on instruction fetch could complicate the design. Multi-ported instruction caches require more energy to operate and need special techniques to handle the cases where instructions are fetched from multiple cache lines. The ability of the IRF to regularly extract multiple instructions from each instruction cache access makes it feasible to reduce the fetch width of the instruction cache, instead relying on frequently executed packed instructions to provide the dispatch stage with a sufficient stream of instructions to avoid starving a wider backend pipeline. This pipeline configuration is shown in Figure 5c.

By packing multiple instruction references into a single MISA instruction, it becomes feasible to decouple the configuration of the fetch pipeline from the design choices of the rest of the pipeline. This increase in the configuration space provides the processor designer greater flexibility in meeting multiple and often divergent, design goals. Use of an IRF in an out-of-order pipeline can also have other positive benefits. The possibility of misspeculation due to overaggressive fetching will be reduced. This occurs because the

Parameter	Low-Power Embedded	High-end Embedded
I-Fetch Queue	4 entries	4/8 entries
Branch Predictor	Bimodal – 128	Bimodal – 2048
Branch Penalty	3 cycles	
Fetch Width	1	1/2/4
Decode Width	1	1/2/3/4
Issue Style	In order	Out of order
Issue Width	1	1/2/3/4
Commit Width	1	1/2/3/4
RUU size	8 entries	16 entries
LSQ size	8 entries	
L1 Data Cache	16 KB 256 lines, 16 B line, 4-way assoc. 1 cycle hit	32 KB 512 lines, 16 B line, 4-way assoc. 1 cycle hit
L1 Instruction Cache	16 KB 256 lines, 16 B line, 4-way assoc. 1/2 cycle hit	32 KB 512 lines, 16 B line, 4-way assoc. 1 cycle hit
L0 Instruction Cache	256 B 32 lines, 8 B line, direct mapped 1 cycle hit	NA
Unified L2 Cache	NA	256 KB 1024 lines, 64 B line, 4-way assoc. 6 cycle hit
Memory Latency	32 cycles	
Integer ALUs	1	1/2/3/4
Integer MUL/DIV	1	1
Memory Ports	1	1/2
FP ALUs	1	1/2/3/4
FP MUL/DIV	1	1
IRF/IMM	4 windows 32-entry IRF (128 total) 32-entry Immediate Table 1 Branch/pack	

Table 1. Experimental Configurations

IRF-resident instructions are those that are most frequently executed, and thus the hot spots of execution will be full of tightly packed instruction sequences. Conversely, the areas that are infrequently executed will tend to exhibit fewer packed instructions and naturally limit the rate of instruction fetch.

5. Experimental Evaluation

This section contains the methods and results used in evaluating the IRF for improving instruction fetch. First, we present our experimental framework and methodology. Second, we combine the IRF with an L0 instruction cache and a 2-cycle non-pipelinable L1 instruction cache to evaluate the impact on energy and execution efficiency. Third, we analyze the energy and execution benefits of decoupling instruction fetch from instruction execution. Finally, we show the overall reduction in static code size due to instruction packing with an instruction register file.

5.1 Compilation and Simulation Framework

All programs are compiled using a modified MIPS port of the VPO compiler [8]. Benchmarks are profiled dynamically and instructions are selected for packing using *irfprof*, a profile-driven IRF selection and layout tool. Each application is then recompiled and instructions are packed based on the supplied layout. The layouts provided for each experiment used four hardware IRF windows for packing [18]. Several IRF-specific optimizations including register renaming, instruction scheduling and instruction selection were performed on all versions of the IRF code to improve code density and performance [19]. Instructions are packed only within the code supplied for each benchmark. Although library code is left unpacked, the dynamic (cycle and energy) results of executing library code are shown in our experimental data in order to provide a thorough evaluation. Packing frequently executed library routines could lead to further improvements in reducing fetch energy and execution time.

We evaluate the performance of each of our machine models using the SimpleScalar simulation environment [3]. Power estimation is performed using Version 1.02 of the Wattch extensions [9] for SimpleScalar. Wattch uses Cacti [34] to model the energy and timing requirements of various pipeline components. Each simulator is instrumented to collect the relevant data involving instruction cache and IRF access during program execution. Energy estimates are based on Wattch's aggressive clock-gating model (*cc3*). Under this scheme, power consumption scales linearly for active units, while inactive portions of the pipeline dissipate only 10% of their maximum power. For our study involving an L0 filter cache, the machine configuration is shown as the *Low-Power Embedded* column in Table 1. Note that 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 to fetch in the embedded processor. We also simulate a 2-cycle non-pipelined L1 instruction cache for several reasons. First, it emulates the worst case performance of an L0 instruction cache (i.e. L0 miss on each instruction fetched from cache). Second, it can be used to demonstrate the benefits of instruction packing for other higher-latency instruction caches that are attempting to reduce power consumption [20], compress executables [10], or encrypt cache contents [31].

The column labeled *High-end Embedded* describes the machine configuration for our second study. In this study, there are no L0 caches, but instead a unified L2 cache for instructions and data. The IRF and IMM configuration remains the same, while most

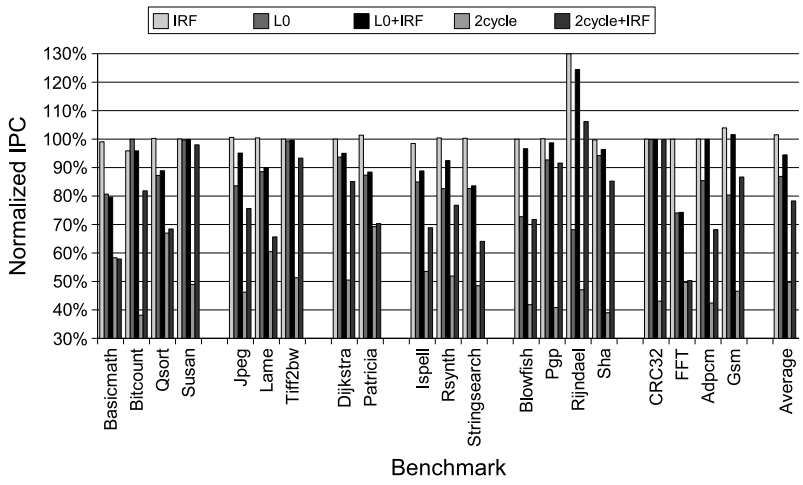

Figure 6. Execution Efficiency with an L0 or 2 cycle Instruction Cache and an IRF

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

Table 2. MiBench Benchmarks

other pipeline structures are scaled up for a higher performance embedded processor. This study features many combinations of fetch, decode, issue and commit widths. The decode, issue, and commit width are always equal to one another, and are always greater than or equal to the fetch width. Fetching is always done using a width that is a power of 2, since we do not want to artificially increase the number of stalls due to instruction mis-alignment. Caches are only able to access a single block in a cycle, so instruction sequences that span multiple cache lines will require several cycles to fetch. Functional units for each of our configurations are added as necessary based on the decode/issue/commit width. The instruction fetch queue is lengthened to 8 entries when the processor reaches a fetch width of 4 instructions.

We use a subset of the MiBench embedded benchmark suite [16] for each experiment. The MiBench suite consists of six categories, each designed to exhibit application characteristics representative of a typical workload in that particular domain. Several of these benchmarks (Jpeg, Gsm, Pgp, Adpcm) are similar benchmarks to those found in the MediaBench suite [23] used in the original evaluation of L0 caches. Table 2 shows the exact benchmarks that were used in the evaluation. For each benchmark with multiple data sets, we selected the small inputs to keep the running time of a full simulation manageable.

5.2 Tolerating L0 Instruction Cache Latency

Each of the graphs in this subsection 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 *Baseline+IRF* corresponds to adding an IRF but no L0 cache. The label *L0* corresponds to adding an L0 instruction cache, but no IRF. *L0+IRF* corresponds to the addition of an L0 instruction cache along with an IRF. With *2cycle*, we are representing a 2-cycle non-pipelinable L1 instruction cache, while *2cycle+IRF* represents this configuration with an IRF. Execution results are presented as normalized instructions per cycle (IPC) as measured by SimpleScalar. Energy results are based on the overall total processor energy consumption.

Figure 6 shows the execution efficiency of various combinations of an L0 or 2 cycle instruction cache and an IRF. Adding an IRF to the baseline processor actually yields a net IPC improvement of 1.04%, primarily due to the code fitting better into the L1 instruction cache. An L0 cache degrades the IPC by 12.89% on average, while adding an IRF cuts this penalty to 6.14%. The 6.75% improvement from adding the IRF to an L0 cache configuration is greater than the original IPC improvement provided by adding the IRF to the baseline processor. These benefits are due to the smaller cache footprint provided by instruction packing, as well as the overlap of fetch stalls with useful RISA instructions. The Rijndael and Gsm benchmarks are even able to surpass the baseline performance with no L0 cache, yielding IPCs of 126.11% and 102.13% respectively. The 2 cycle implementation reduces IPC to 49.73% of its original value, while adding an IRF increases IPC to 78.28%, nearly a 57% improvement. Several of the automotive category benchmarks experience diminished performance when adding an IRF, due to their dependence on library routines which are not packed. However, it is clear that the IRF can be used to mask some of the performance issues involved in complex fetch configurations that involve low-power components, compression or even encryption.

The energy efficiency of the varied IRF, L0 and 2 cycle instruction cache configurations are shown in Figure 7. Adding an IRF to the baseline processor yields an average energy reduction of 15.66%. The L0 cache obtains an overall reduction of 18.52%, while adding the IRF improves the energy reduction to 25.65%. The additional energy savings in the final case are due to two factors. First, fetching instructions via the IRF requires less energy than fetching from the L0 instruction cache, and second, the ability of the IRF to tolerate L0 cache misses improves the overall execution time of the application. When dealing with the 2 cycle instruction cache, the overall energy is increased by 17.63%. It is important to note that the Wattch model places a heavy emphasis on the clock gating of unused pipeline units, hence the reason for the energy consumption only increasing by 5-25% for the extended running times. Adding an IRF to this configuration, however, re-

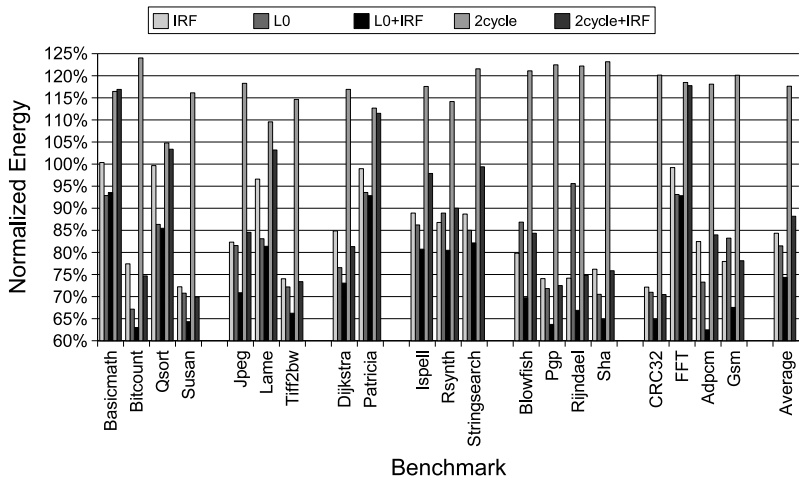

Figure 7. Energy Efficiency with an L0 or 2 cycle Instruction Cache and an IRF

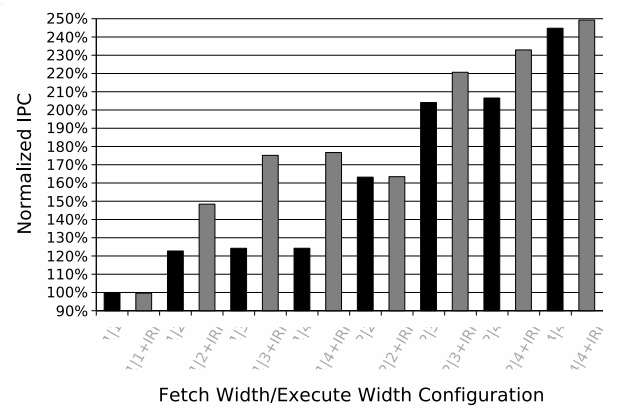

Figure 8. Execution Efficiency for Asymmetric Pipeline Bandwidth

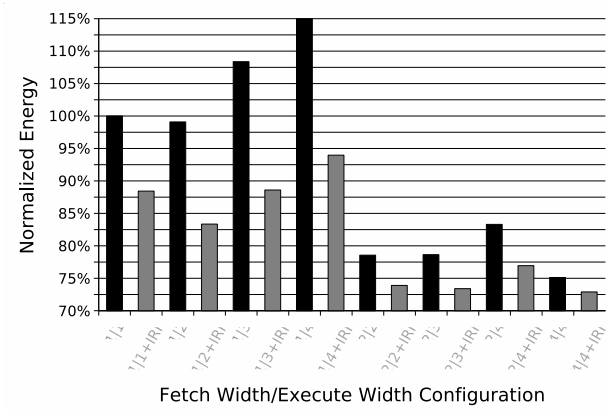

Figure 9. Energy Efficiency for Asymmetric Pipeline Bandwidth

duces the overall energy consumption to 88.19% of the baseline, which is only slightly higher than the configuration of a 1-cycle L1 instruction cache with IRF.

The fetch energy characteristics of the IRF and L0 instruction cache are also very interesting. The baseline processor fetch energy consumption can be reduced by 33.57% with the introduction of an IRF. An L0 instruction cache instead reduces the fetch energy consumption by 60.61%, while adding both yields an overall fetch energy reduction of 70.71%. This savings can be extremely beneficial for embedded systems that have a significant portion of their total processor energy expended during instruction fetch.

5.3 Effectively Utilizing Asymmetric Pipeline Bandwidth

Each of the graphs in this subsection use the following conventions. All results are normalized to the baseline processor model, which has a single instruction fetch, decode, issue and commit bandwidth. This is the leftmost bar presented in each of the graphs. The graph is plotted linearly by instruction fetch width, followed by the execute width (decode/issue/functional units/commit width) as a tie-breaker. Black bars are used to denote configurations that do not have an IRF, while gray bars indicate the presence of an IRF.

Figure 8 shows the normalized IPC over all benchmarks for the various configurations of the fetch and execution engine. The configurations without an IRF are often able to make some use of the additional functional unit bandwidth (1/1 to 1/2, 2/2 to 2/3), however they plateau quickly and are unable to use any additional available bandwidth (1/2 through 1/4, 2/3 to 2/4). The IRF versions, alternately, are able to improve steadily compared to the previous IRF configuration, except in the 1/4+IRF case. This configuration is overly aggressive, as the IRF is unable to supply 4 instructions per cycle for execution. Branches and other parameterized RISA instructions occupy two slots in the packed instruction format, thus limiting the maximum number of instructions fetchable via a single MISA instruction. Additionally, the IRF selection heuristic is a greedy one, and we will never be able to produce all tightly packed instructions with 4–5 slots completely full. This limits even the IRF's ability to supply instructions to this overly aggressive backend, but additional fetch bandwidth more than makes up for this limitation in more complex configurations (2/3+IRF, 2/4+IRF). Overall however, the IRF configurations have an average IPC that is 15.82% greater than corresponding configurations without an IRF.

The energy efficiency of the various configurations of asymmetric pipeline bandwidth are shown in Figure 9. Results are presented

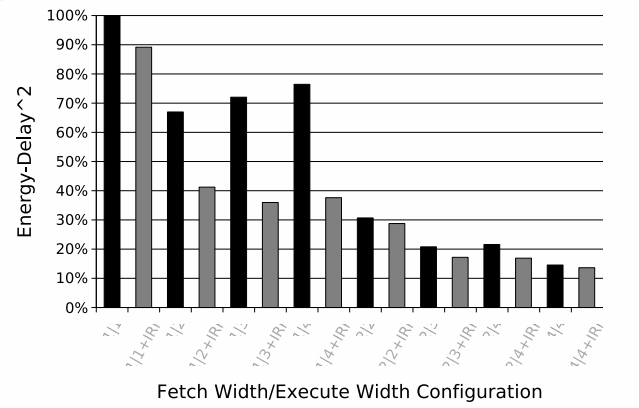

Figure 10. Energy-Delay^2 for Asymmetric Pipeline Bandwidth

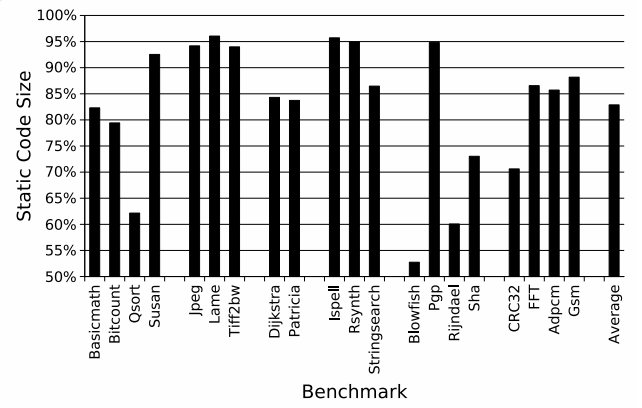

Figure 11. Reducing Static Code Size with an IRF

as normalized energy to the 1/1 baseline processor configuration. Note that the Wattch model places a heavy emphasis on the clock gating of unused pipeline units. Since the running time of the more complex configurations is lower, the total energy is correspondingly reduced as compared to the baseline processor. The results show that increasing the execution width by one additional instruction can yield more efficient energy usage (1/1 to 1/2, 1/1+IRF to 1/2+IRF, 2/2+IRF to 2/3+IRF), but greater increases in all other cases are not beneficial. Overall though, the energy results show that IRF configurations can be as much as 22% more energy efficient (1/3+IRF, 1/4+IRF) than corresponding configurations without an IRF. On average, the IRF configurations utilize 12.67% less energy than corresponding non-IRF configurations.

Figure 10 shows the energy-delay squared (ED^2) for the various asymmetric fetch and execution bandwidth configurations. Energy-delay squared is an important metric for exploring the processor design space. As expected, this graph is similar to the IPC results shown in Figure 8, as the configurations without an IRF continue to have problems with keeping the additional functional units of the pipeline busy (1/2 through 1/4, 2/3 to 2/4). The IRF configurations see a small increase (1.64%) in ED^2, when moving from a 1/3+IRF to 1/4+IRF, primarily due to the increased energy requirements of this configuration. Without this data point, however, the IRF configurations smoothly scale for the varied combinations of fetch and execute bandwidth available in the processor pipeline. On average, the relative decrease in ED^2 for using the IRF configurations over the non-IRF versions is 25.21%. In addition, by decoupling instruction fetch and instruction execution, a computer architect has greater freedom to explore an increased number of processor configurations with a wider variety of energy/execution characteristics.

5.4 Reducing Static Code Size

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 used in our experiments. 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 executable size by 17.12% 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 be placed in the IRF and referenced in various combinations to construct the necessary functionality for each similar routine. The consumer and office benchmarks are not as easily packed, since they often consist of many routines that are accessed infrequently during normal usage. However, they are still able to achieve a significant level of compression due to packing with multiple windows.

6. 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 [6, 7]. 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 L-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 besides the loop branch [24]. Instructions 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 [25]. Another improvement to the loop cache is the use of preloading and hybridization [15]. 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%. A previous study has also shown that an IRF can interact very favorably with

a loop cache, providing fetch energy reductions that neither could achieve separately [18].

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 [32]. 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 [12]. 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.

One power-saving method that has been successfully applied in the area of high-performance processors is the concept of pipeline gating or front-end throttling [28]. These techniques reduce the energy expenditure of a processor by actively suppressing the fetch of wrong-path instructions. When a low-confidence branch is encountered in the pipeline, instruction fetch and potentially instruction decode are stalled for a number of cycles. Energy is saved by not having to fetch, decode and eventually flush these potential wrong-path instructions. Many heuristics have been developed using branch confidence [2], and instruction flow metrics [4] to determine the proper times for throttling back the fetch bandwidth. Sherwood et al. improve throttling selection by dynamically profiling regions at run-time [30]. Annavaram et al. have similarly proposed an EPI (energy per instruction) throttling technique for reducing energy consumption during periods of low ILP [1].

7. Future Work

There exist many avenues suitable 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 instruction packing combined with other architectural features that affect fetch latency. For instance, the IRF can be integrated with a more energy efficient pipeline backend using asymmetric-frequency clustering [5], which attempts to distribute instructions to execution engines with varying performance/energy characteristics. Packed instruction density could also be used to differentiate code sequences with varying levels of ILP, by adjusting the criteria for IRF selection and packing.

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 [14, 11, 10], and echo factoring [22] 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.

8. Conclusion

In this paper we examined how an instruction register file incorporated into an instruction set architecture interacts with micro-architectural components in both low-power and high-performance embedded processors. In addition, the use of instruction packing, which is a fundamental component of the IRF micro-architecture, allows for significant reductions in the overall static code size of an application.

When energy is the overriding design parameter, we have evaluated the interactions between a small low-power L0 instruction cache and an IRF. 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, increasing the overall IPC by 6.75% on average. Additionally, both the L0 cache and the IRF can interact such that the total energy consumption is further reduced by an additional 5.78% on average.

For high-performance embedded systems, the introduction of the IRF can reduce pressure on instruction fetch by enabling multiple instructions to be referenced in each packed instruction fetched from the instruction cache. This results in a performance boost in those portions of the program where performance is limited by fetch bandwidth, such as when recovering from a branch misprediction and needing to quickly fill the reorder buffer to extract more instruction level parallelism. Our results show that adding an IRF can improve IPC and energy consumption not only for single issue architectures, but also for multi-issue designs by an average of 15.82% and 12.67%, respectively. The use of the IRF also gives the processor architect the ability to scale back the fetch width in order to meet tighter power constraints, while relying on frequently packed instructions to feed a more aggressive back-end pipeline. This provides greater flexibility in the overall pipeline design to meet all design goals.

We showed that it is possible to improve performance, reduce code size and reduce energy consumption by adding an IRF to an existing architecture. These improvements are found at both ends of the processor design spectrum — low-power single-issue in-order and high-performance multiple-issue out-of-order embedded processors. Thus, the IRF serves to decouple instruction cache access from the processor backend, making both low-power and high-performance embedded processors more efficient.

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, CCF-0444207, and CNS-0615085.

References

[1] ANNAVARAM, M., GROCHOWSKI, E., AND SHEN, J. Mitigating amdahl's law through epi throttling. In *Proceedings of the 2005 ACM/IEEE International Symposium on Computer Architecture* (2005), IEEE Computer Society, pp. 298–309.

[2] ARAGÓN, J. L., GONZÁLEZ, J., AND GONZÁLEZ, A. Power-aware control speculation through selective throttling. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 103–112.

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

[4] BANIASADI, A., AND MOSHOVOS, A. Instruction flow-based front-end throttling for power-aware high-performance processors. In *ISLPED '01: Proceedings of the 2001 international symposium on*

Low power electronics and design (New York, NY, USA, 2001), ACM Press, pp. 16–21.

[5] BANIASADI, A., AND MOSHOVOS, A. Asymmetric-frequency clustering: a power-aware back-end for high-performance processors. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design* (New York, NY, USA, 2002), ACM Press, pp. 255–258.

[6] 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.

[7] 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.

[8] 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.

[9] BROOKS, D., TIWARI, V., AND MARTONOSI, M. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer architecture* (New York, NY, USA, 2000), ACM Press, pp. 83–94.

[10] 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.

[11] 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.

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

[13] FOLEGNANI, D., AND GONZÁLEZ, A. Energy-effective issue logic. In *Proceedings of the 28th annual International Symposium on Computer architecture* (New York, NY, USA, 2001), ACM Press, pp. 230–239.

[14] 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.

[15] 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.

[16] 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).

[17] 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.

[18] 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* (November 2005), IEEE Computer Society, pp. 19–29.

[19] HINES, S., WHALLEY, D., AND TYSON, G. Adapting compilation techniques to enhance the packing of instructions into registers. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (October 2006), pp. 43–53.

[20] KIM, N. S., FLAUTNER, K., BLAAUW, D., AND MUDGE, T. Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 219–230.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] LEFURGY, C., BIRD, P., CHEN, I.-C., AND MUDGE, T. Improving code density using compression techniques. In *Proceedings of the 1997 International Symposium on Microarchitecture* (December 1997), pp. 194–203.

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

[28] MANNE, S., KLAUSER, A., AND GRUNWALD, D. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 1998 ACM/IEEE International Symposium on Computer Architecture* (1998), IEEE Computer Society, pp. 132–141.

[29] 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.

[30] SHERWOOD, T., SAIR, S., AND CALDER, B. Phase tracking and prediction. *SIGARCH Comput. Archit. News 31*, 2 (2003), 336–349.

[31] SHI, W., LEE, H.-H. S., GHOSH, M., LU, C., AND BOLDYREVA, A. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 2005 ACM/IEEE International Symposium on Computer Architecture* (2005), IEEE Computer Society, pp. 14–24.

[32] 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.

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

[34] WILTON, S. J., AND JOUPPI, N. P. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid State Circuits 31*, 5 (May 1996), 677–688.