Reducing Instruction Fetch Cost by Packing Instructions into Register Windows

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

Abstract

Instruction packing is a combination compiler/architectural approach that allows for decreased code size, reduced power consumption and improved performance. The packing is obtained by placing frequently occurring instructions into an Instruction Register File (IRF). Multiple IRF entries can then be accessed using special packed instructions. Previous IRF efforts focused on using a single 32-entry register file for the duration of an application. This paper presents software and hardware extensions to the IRF supporting multiple instruction register windows to allow a greater number of relevant instructions to be available for packing in each function. Windows are shared among similar functions to reduce the overall costs involved in such an approach. The results indicate that significant improvements in instruction fetch cost can be obtained by using this simple architectural enhancement. We also show that using an IRF with a loop cache, which is also used to reduce energy consumption, results in much less energy consumption than using either feature in isolation.

1 Introduction

One important design constraint for many embedded systems is power consumption, which can affect battery life and operating temperature. Reducing the power consumption associated with I-fetch logic, which consumes approximately one third of the total processor power [18], is a natural target for making embedded processors more energy efficient. There have been a number of different approaches to reduce the power consumption associated with I-fetch logic. L0 caches can significantly reduce power consumption, but incur an associated execution time penalty [13]. Zero overhead loop buffers [7] and loop caches [9] can also reduce fetch cost, but only for a specific class of the innermost loops in a function.

An orthogonal approach has been recently proposed that places instructions into registers, enabling the compiler to *pack* the most frequently executing instructions into an Instruction Register File (IRF) [11]. This approach allows multiple instructions to be specified (by IRF index) in a single instruction fetched from the instruction cache (IC). Using an IRF not only saves energy by reducing the number of instructions referenced from the IC, but also saves space by reducing the number of instructions stored in memory, and can improve execution time by streamlining instruction fetch and improving the IC hit rate.

The previous work on investigating the benefits of an IRF focused on using a single 32-entry IRF containing the most frequently executed instructions for an application. The IRF was initialized with the selected instructions at the time the application was loaded and remained unchanged for the lifetime of the application. Even with this very restrictive IRF allocation strategy, the results showed significant energy savings, particularly for smaller applications.

In this paper we investigate IRF allocation policies that enable the compiler to select those instructions most useful for each phase of execution [20]. This approach reduces energy consumption by providing more instructions to the processor pipeline from the IRF. We develop two heuristics for determining what instructions will be placed into the IRF and at what points in the program the instructions are promoted to the IRF. We also propose microarchitectural enhancements to minimize the overhead in changing IRF contents during execution. Finally, we also evaluate how the use of an IRF interacts with another instruction fetch energy reduction technique — the loop cache.

The remainder of this paper has the following organization. First, we review previous work on reducing instruction fetch cost, including the prior work on packing instructions into registers. Second, we describe our IRF framework and the model used for evaluating its performance. Third, we develop a more sophisticated IRF allocation heuristic for modifying the contents of the IRF during execution, and evaluate the resulting reduction in energy as well as the

overhead of inserting instructions to load the IRF at various points in the program. Fourth, we develop a new IRF microarchitecture that reduces the overhead required to modify the IRF instructions during execution, providing greater energy reduction benefits when compared to the original IRF with compiler instruction promotion only. Fifth, we show that the total number of IRF registers can be reduced with only a slight increase in fetch cost by having a portion of the visible entries in the IRF contain a fixed set of instructions. Sixth, we illustrate that a loop cache and the IRF can be used together to further significantly reduce instruction fetch cost. Seventh, we present some crosscutting issues regarding the support costs of using a windowed approach for the IRF. Eighth, we mention several potential topics for future work. Finally, we present our conclusions for the paper.

2 Previous Work on Reducing Instruction Fetch Cost

There have been a number of approaches used to reduce instruction fetch cost. They can be broadly placed into three groups: code compression, alternate instruction caching strategies, and alternate instruction storage strategies with compiler selection.

The first approach is to compress the code within a program, which can have the side effect of decreasing the number of cache misses due to a smaller footprint of instructions being accessed during the execution. One technique is to abstract common code sequences into routines and have the original sites of each sequence converted into calls [8, 6, 4]. A hardware extension of this approach is to use an *echo* instruction, which indicates where the instruction sequence can be found and the number of instructions to be executed [15]. This approach eliminates the need for an explicit return instruction at the end of the abstracted sequence and allows abstracted sequences of instructions to overlap. However, both procedural abstraction and echo factoring can increase execution time due to the extra transfers of control and can degrade instruction cache performance due to diminished spatial locality.

Other techniques use a hardware dictionary, where duplicate code sequences are placed in a special control store in the processor and codewords are associated with each of these sequences [17, 5]. These approaches have the disadvantage of complicating instruction fetch and decode logic since instructions can now vary in size. Yet another technique is to use dual instruction sets, which support both 16-bit and 32-bit instruction formats [19, 14]. While using the 16-bit mode typically saves space, this occurs at a cost of additional instructions and increased execution times. More recently, Cheng, et al. proposed a new 16-bit ISA that avoids some of the instruction overhead found in other 16-bit ISAs by replacing the instruction decoder with a programmable decoder that enables a subset of the instructions implemented in the microarchitecture to be mapped to the ISA [3].

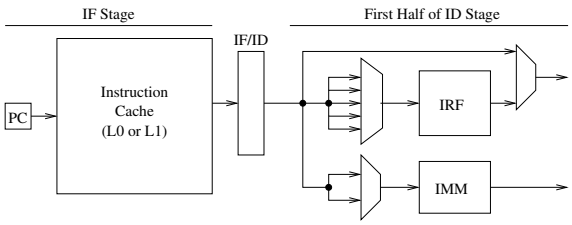

Figure 1. Decoding a Packed Instruction

There have also been hardware features explicitly developed to reduce energy consumption associated with instruction fetches. One technique is to use a zero overhead loop buffer, where an innermost loop is explicitly loaded and executed [7]. Such loops must typically be limited in that the number of instructions must fit into the buffer, there can be no transfers of control besides the loop branch, and the number of iterations to be executed must be known. Another technique is to use an L0 cache, which is very small and requires much less energy to access than an L1 instruction cache [13]. However, L0 caches have high miss rates, which increase execution times. Finally, loop caches have been used that can be dynamically loaded when short offset backward branches are encountered [9]. The set of loops that can be cached using this approach is limited to those that have a small number of instructions and no transfers of control occur besides the branch back to the top of the loop.

The work in our 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. 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. Figure 1 shows the use of an IRF at the start of the instruction decode stage. Figure 2 shows the special MISA instruction format used to reference multiple instructions in the IRF. These instructions are called *packed* since multiple RISA instructions are referenced in 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 and the 1-bit S field. The compiler uses a profiling pass to determine the most frequently referenced instructions that

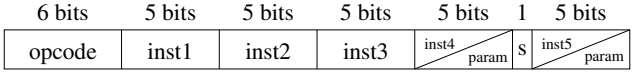

Figure 2. Tightly Packed Format

Table 1. 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

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 to the RISA *nop* terminates execution of the packed MISA instruction so no performance penalty is incurred. The compiler uses a second pass to pack the MISA instructions into the tightly packed format shown in Figure 2.

3 Evaluation of IRF Enhancements

This section provides an overview of the IRF framework used to carry out the experiments described in this paper. 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. Our baseline IRF model has 32 instruction register entries and supports parameterization via a 32-entry immediate table. For all of our experiments, only the size of the IRF is increased as preliminary studies showed little to no benefit for varying the IMM size.

Code is generated using a modified port of the VPO compiler for the MIPS [2]. Each benchmark is profiled dynamically using SimpleScalar and then instructions are selected for packing using *irfprof*, our profile-driven IRF selection and layout tool. The application is then recompiled and instructions are packed based on the IRF layout provided by *irfprof*. Instruction packing is performed only on the code generated for the actual source provided for each benchmark. Library code is left unmodified and as such is not evaluated in our results.

We chose to use the MiBench embedded benchmark suite for our experiments [10]. MiBench consists of six categories, each designed to exhibit application characteristics representative of a typical embedded workload in that particular domain. Table 1 shows the benchmarks evaluated in each of our experiments, although some of the figures will only show category averages to save space.

The IRF simulator includes a power analysis estimation based on and validated using sim-panalyzer [21]. It calculates approximations of area size and number of gates for the IC, IRF, immediate table and combinational logic associated with instruction fetch (there are negligible effects on the rest of the pipeline). In fact, for a fixed, 8K-byte IC and 32-entry IRF, the energy requirements for each configuration can be very accurately estimated with the following equation where Cost_{IRF} is approximately two orders of magnitude less expensive than Cost_{IC}:

$$\begin{aligned} \text{E}_{\text{fetch}} \quad = \quad & \text{Cost}_{\text{IC}} \times \text{Accesses}_{\text{IC}} + \\ & \text{Cost}_{\text{IRF}} \times \text{Accesses}_{\text{IRF}} \end{aligned}$$

Actual parameters were calculated using the Cacti cache modeling tool [24] with extensions for handling the IRF. As IRF sizes are increased, Cost_{IRF} increases in a similar fashion as with traditional register file implementations. These estimates are conservative for the IRF, considering that a windowed approach can take advantage of drowsy cache techniques [12] for powering down unused windows, which was not modeled in our experiments.

4 Dynamic IRF Recomposition via Software Windowing

In prior work on instruction packing with an IRF, the IRF-resident instructions were selected using a profiled run of the application and assigned for the duration of the application's execution. The most frequently fetched instructions were placed in the IRF and *packed* instructions were inserted into the application code referencing sequences of IRF instructions. Since the instructions in the IRF were unchanged by the application during execution, the benefits of IRF could be reduced if the application has phases of execution with radically different instruction composition. No individual phase will be able to obtain maximum instruction packing with a single statically loaded IRF, since only the most frequently executed instructions from each phase will exist in the IRF. A natural extension to improve IRF utilization is to enable the modification of instructions in the IRF during program execution. This will enable a better selection of instructions for each phase of the program execution. Our first approach replaces the load-time allocation of instructions into the IRF with compiler managed allocation throughout execution by including *load_irf* instructions. This approach enables the compiler to manage the IRF storage in a manner similar to data register promotion. The IRF contents at a particular point in execution are viewed as an IRF window. Each function could logically have its most frequently executed instructions in its unique IRF window. These windows would likely be shared among functions to reduce the switching necessary during calls and returns.

Functions are natural entities to allocate to windows since they are compiled and analyzed separately. The prior work on exploiting instruction registers used the entire execution profile to decide which instructions would be allocated to the IRF for the entire execution. In effect, one can view the prior work as allocating all of the functions to a single window. When multiple windows are available, we must decide how to partition the set of functions in a program among these windows. The goal is to allocate functions whose set of executed instructions are similar to the same window so that the windows can be best exploited.

Instructions can be inserted that load an individual IRF entry from memory. Thus, there will be a cost associated with switching IRF windows. To assess how effective this approach can be, we devised an algorithm that takes into account the cost of switching the instructions available via the IRF. This algorithm is shown in Figure 3. We first profiled each application to build a call graph with edges weighted by the total number of calls between each pair of functions. Initially all functions start as part of a single IRF window. We take a greedy approach to adding partitions, selecting the most beneficial function to be either placed in a new partition of its own or merged with another existing partition. The goal of this algorithm is to keep functions in the same window unless the benefit for switching windows outweighs the cost of additional instructions to load and restore IRF windows. Each time we calculate the cost of allocating a function to a specific window, we also include the switching cost if the function being allocated either invokes or is invoked by a function in a different window. We determine the difference in IRF entries between the two windows and assess the cost of loading the instructions for the new window at the point of the call and loading the instructions for the old window at the point of the return. One can view this cost as a lower bound since it may be difficult to keep the common instructions in the same IRF entry locations since a function may be invoked by many other functions using different windows. Additionally, we only place each function in a partition at most one time in order to reduce the partitioning overhead. Once the partitions have been constructed, the compiler determines which instructions can be packed and at what call sites IRF loads must occur to maintain the correct contents of the IRF.

We evaluate the effectiveness of applying a software partitioning against the single window IRF partitioning. Figure 4 shows the results for performing software partitioning on each benchmark. Each benchmark is shown individually along with the averages for each category and the entire suite. The software window approach obtains an average fetch cost of 54.40% compared to not using an IRF, while the original single IRF only obtains 58.08%. Several of the benchmarks perform worse after packing instructions with software windows. This is due to the heuristic used for

```
Read in instruction profile for each function;
Read in callgraph along with estimated edge weights;
Merge all functions into a single IRF window;
changes = TRUE;
while changes do
    changes = FALSE;
    best_savings = 0;
    split_type = NONE;
    foreach function i that has not been placed do
        new_savings = benefit of placing i in a new window;
        // 1) Find maximum benefit split,
        if new_savings > best_savings then
            best_savings = new_savings;
            split_function = i;
            split_type = NEW_WINDOW;
    foreach function i that has not been placed do
        foreach window k from 1 to n-1 do
            new_savings = benefit of placing i in window k;
            // 2) Find maximum benefit merge
            if new_savings > best_savings then
                best_savings = new_savings;
                split_function = i;
                split_window = k;
                split_type = ADD_TO_WINDOW;
    if best_savings > 0 then
        if split_type == NEW_WINDOW then
            create a new window n and move split_function into it;
            // 3a) Perform the split,
            n += 1;
        else
            //split_type == ADD_TO_WINDOW
            move split_function from its original window into
            split_window;
            // 3b) Or perform the merge
        changes = TRUE;
        mark split_function as placed;
```

Figure 3: Partitioning Functions for Software IRF Windows

choosing which instructions to pack. In the single IRF case, only the extremely frequent instructions from tight loops become packed, leading to a great initial savings. This is particularly true for the applications with a few dominant loops. The greedy heuristic used to partition the functions can underestimate the overhead of a function removed from the primary partition early in the processing because it assumes that unprocessed functions will not increase the overhead. Of course some of these later functions will be placed in different partitions increasing the overhead. The smaller the benefit of partitioning, as seen in some of the smaller applications, the more likely inefficiencies in the heuristic will negate the advantage of performing IRF partitioning. The larger benchmarks see marked improvement (up to 31% savings with Lame), as they are more likely to exhibit different phases of execution and can better partition their phase behavior using multiple IRF windows.

The number of software partitions allocated for each benchmark ranges from 1 through 32 (Pgp) with a median of only 4 and a mean of 8.33. The relatively small number of partitions reduces the overhead of loading instructions into the IRF by reducing the number of function calls that

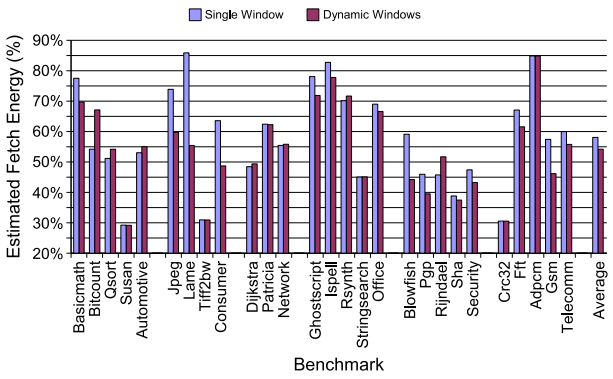

Figure 4. Reducing Fetch Energy by Software Partitioning IRF Windows

require a new set of instructions placed into the IRF. Each call site may also only change a subset of the IRF entries since the IRF windows may contain some of the same instructions. In fact, the average number of IRF entries that differ between partitions is 24.54, so the overhead for each function call between different IRF partitions averages 49 new instructions — half each for allocating the new instruction and restoring at the function return. The small number of partitions also allows us to consider a hardware solution that can eliminate most of the overhead of loading a single IRF in favor of providing multiple IRF windows in the microarchitecture.

5 Partitioning Functions into Hardware Windows

In the previous section, we found that applying software windowing techniques to the IRF can lead to reductions in fetch cost, particularly for large programs with varying phase behaviors. One of the problems for smaller programs however is the added overhead of switching IRF windows in software. In addition, the software partitioning approach requires an explicit call graph. This approach does not enable functions called through pointers to be effectively partitioned since this would make it difficult, if not impossible, to determine which IRF entries must be loaded. In this section we evaluate an IRF design supporting several IRF windows, which can be filled with instructions at load time, thus eliminating the overhead of switching windows. Since we know that most programs use only a small number of partitions anyway, it is feasible to add a limited number of windows into the microarchitecture and still see the power savings — without the overhead.

Register windows have been used to avoid saving and restoring data registers. On the SPARC architecture, the

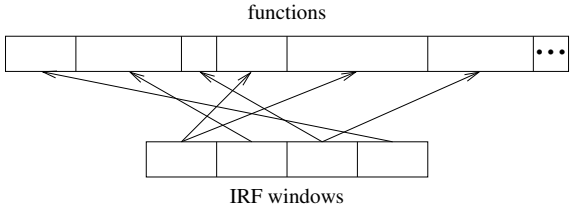

Figure 5. Example of Each Function Being Allocated an IRF Window

windows are arranged in a circular buffer [23]. A *register window pointer* (RWP) is used to indicate the current window that is to be accessed on each register reference. The RWP is advanced one window when entering a function and is backed up one window when leaving a function. When the RWP is advanced onto an active window (an overflow), then an exception occurs and the window's contents are spilled to memory. When the RWP is backed up onto a window that does not contain the correct contents (an underflow), then an exception occurs and the window's contents are loaded from memory. A circular buffer is an appropriate model for holding these data values since the data registers are used to contain values associated with function activation records, which are allocated in a LIFO manner.

Unlike data values associated with function activation records, the instructions associated with each function are fixed at compile time. Thus, we modified our compiler to statically allocate each function to a specific window. Figure 5 depicts both functions and the IRF windows associated with them. While the functions can vary in the type of instructions and their frequency of execution, the IRF windows all have the same size. The drawback of raising exceptions due to overflowing and underflowing data windows on the SPARC does not occur in our design due to each function being statically allocated to an instruction window.

In order to reduce the overhead of switching from one window to another, we encode the window to be used by the called function as part of the call instruction's target address. The MIPS call instruction uses the J format shown at the top of Figure 6. The modified call instruction uses the same format, but the high order bits are now used to set the instruction RWP (IRWP) and only the low order bits are used to update the program counter. While this change does decrease the maximum size of programs from 2^{26} instructions, even retaining only 20 bits for the actual target address would allow over two million (now more compressed) instructions, which should be more than enough for almost all embedded applications. Rather than just saving the return address in a data register ($31 on the MIPS), the call instruction also saves the value of the current IRWP. Thus, the semantics of the return instruction, `jr $31` on the MIPS,

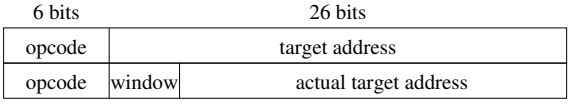

Figure 6. Encoding the IRF Window Number As Part of the Call Target Address

is also changed to reset the IRWP. Thus, the processor has an entire cycle during each call and return instruction to set the IRWP. Note that calls through pointers can also be handled since the linker can modify the address associated with a function so that the high order (or low order) bits indicate which window is associated with the function.

Rather than having to go through an IRWP for every access to an entry in the IRF, the hardware could alternatively copy the registers each time the window is changed during the execution of a call or return instruction. Parallel register moves can be performed between the IRF and the appropriate window, which is similar to the boosting approach used to support speculation by copying a number of shadow registers to general-purpose registers in parallel [22].

Register windows are a more attractive approach than just increasing the number of total available registers for hardware such as the IRF. First, increasing the number of registers in the IRF without windowing would force the RISA instruction specifiers to grow beyond the 5 bits they currently occupy. Moving to 64 entries would require 6 bits, from which it would not be possible to pack 5 entries together ($6 + 5 \times 6 = 36$ bits > 32 bits). Thus, at most 4 RISA instructions could exist in a tightly packed instruction, limiting the number of pure IRF fetches that can possibly be attained. Prior research shows that 13.35% of the dynamic instructions fetched from the IC are these very tightly packed instructions [11], so cutting them down to four entries might drastically affect the fetch performance. Second, the larger IRF would also consume more power as the entirety would need to be active on each instruction decode. With windowing, some portions can be placed in a low-power state when they are not being actively used.

Figure 7 depicts the algorithm that we used to perform this partitioning. The primary difference between this heuristic and the previous software partitioning heuristic is that this version does not need to estimate overhead costs. The algorithm begins by reading in the instruction profile that indicates how often each type of instruction is executed for each function. We are concerned with the *type* of instruction since we are able to parameterize immediate values so that several distinct instructions can refer to the same RISA instruction. The algorithm then estimates a cost for each function, where the 31 most frequently executed types of instructions have a fetch cost of 1 and the remaining instructions have a fetch cost 100 times greater, which serves

```
Read in instruction profile for each function;
Select the function with the highest estimated cost and assign it to
window 0;
m = 1;
while m < number of IRF windows do
    max_func_increase = 0;
    // Initially allocate most beneficial
    // function for each partition
    foreach function i not yet assigned do
        if max_func_increase < worst-case cost of function i then
            min_increase = worst-case cost of function i;
            foreach window of the m current windows do
                old_cost = current cost associated with window;
                new_cost = cost associated with current window
                  along with merging function i into this window;
                if new_cost - old_cost < min_increase then
                    min_increase = new_cost - old_cost;
            if max_func_increase < min_increase then
                max_func_increase = min_increase;
                k = i;
    Assign the function k to the new window m;
    m += 1;
Sort the functions according to their worst-case cost;
// Allocate remaining functions
foreach function not yet assigned do
    Assign the function to the window that results in the lowest
      increase in cost;
```

Figure 7: Partitioning Functions for Hardware IRF Windows

as a simple estimate for the relative difference in energy consumption from fetching an instruction from the IRF versus the IC. The 31 most frequently executed types of instructions are selected since one of the 32 entries in the IRF has to be reserved to represent a *nop* (no operation) when not all of the RISA instruction fields in a packed instruction can be filled, allowing a portion of the fields to be used without wasting processor cycles. For each of the remaining windows the algorithm determines for each function the minimum increase in cost that would be required to allocate that function among the currently assigned windows. The algorithm allocates the next window to the function that would cause the greatest minimum increase in cost. In other words, we initially allocate a single function to each window taking into account both the estimated cost of the function and overlap in cost with the functions allocated to the other windows. The worst-case cost is used to improve the efficiency of the algorithm. For the worst-case cost we assume that none of the instructions in the functions are allocated to the IRF. The final pass is used to assign each of the remaining functions to the window that results in the lowest increase in cost. We sort the functions so that the functions with the greatest number of executed instructions are assigned first. The rationale for processing the functions in this order is to prioritize the allocation of functions that will have the greatest potential impact on fetch cost.

For the hardware partitioning scheme, we vary the number of available hardware windows from 1 through 16. Each

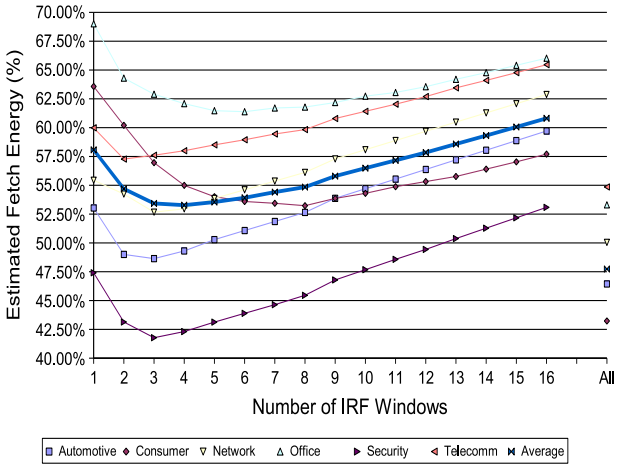

Figure 8. Reducing Fetch Energy by Varying the Number of Hardware Windows

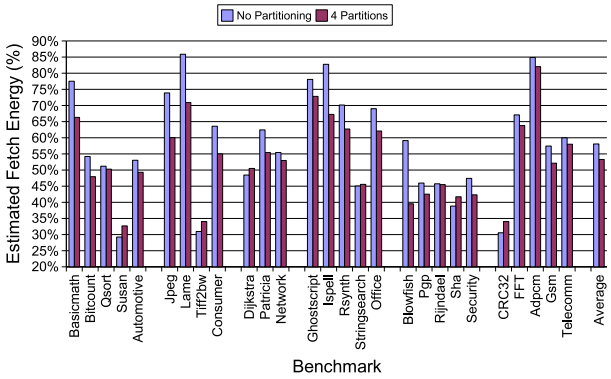

Figure 9. Reducing Fetch Energy with a 4 Partition IRF

time the number of windows is increased, we recalculate the power statistics for the IRF based on the number of new instruction registers. We also evaluate an ideal scheme (all) that produces a window for each individual function while using the power statistics for a single window. Due to the large number of program runs, the results are presented by benchmark category in Figure 8. The original IRF scheme (one window) achieves an average 58.08% fetch cost, while moving to two windows shrinks the average fetch cost to 54.69%. The ideal case only provides an additional 5% savings in fetch cost on average over four windows, which provides a fetch cost of 53.28%. After four windows, however the average results show slight increases in instruction fetch energy, since the larger IRFs require an increase in the energy required to access and store the frequent instructions. Since we can pack no more than 5 RISA instructions in each 32-bit MISA instruction the theoretical limit to fetch cost is approximately 21.5% based on using the power statistics for a 32-entry IRF. However, some restrictions are placed on where instructions can be located in a packed instruction (e.g. branches and branch targets cannot be placed in the middle of a pack), so the best results fall short of the theoretical limits even for very large IRF window sizes. However, the average improvements exceed those for the compiler managed IRF because the overhead has been almost entirely eliminated.

Figure 9 depicts a slice of the results from Figure 8, detailing the per benchmark results for partitioning with four hardware windows. The graph shows an IRF with only a single window, as well as the four window IRF compared to a baseline fetch unit with no IRF. Although a few of the benchmarks see a small increase in fetch energy when moving to four windows, there is still a substantial overall fetch energy savings, and the additional partitions can be used to improve more phase-diverse benchmarks like Jpeg and Blowfish. Many of the benchmarks are able to obtain a large benefit from the hardware IRF windows since they have a diverse instruction composition among several frequently accessed functions. The hardware partitioning allows function call and return switching costs to be eliminated in these cases, leading to improved instruction packing and reduced fetch cost. However, a few of the applications (Lame) do not achieve the same savings as the compiler managed IRF when the number of IRF windows is smaller than the partitions used in the results presented in Section 4.

6 Reserving a Portion of the IRF to be Static

In the prior work on packing instructions into an IRF, the IRF was statically assigned instructions that would remain the same for the entire execution of each application [11]. For the windowed IRF approach, we found that often the same instructions were stored in different windows. This fact leads us to believe that some instructions may be common enough to warrant packing for the entire application execution. In this section we explore keeping a portion of the IRF fixed throughout the entire execution and allowing the remaining portion to change as calls to and returns from functions in different IRF windows occur. This concept is similar to the SPARC data registers, where `r0-r7` of the visible register set correspond to global registers which never change and `r8-r31` correspond to windows within the register file [23]. The general idea is that if a small set of the most frequently executed instructions types can be used for all of the functions, then the remaining portion of the visible register set can refer to smaller windows and the total size of the IRF can be decreased with only a very small impact on packing instructions. The benefits of a smaller

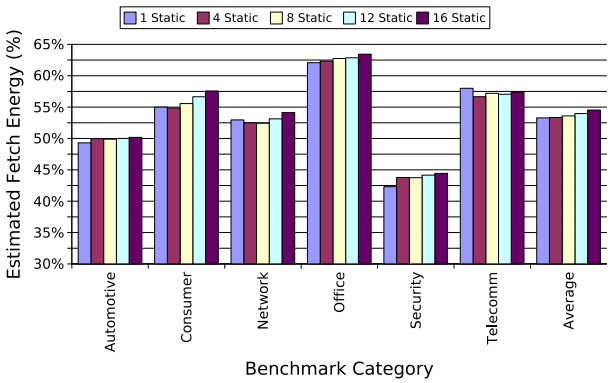

Figure 10. Reserving a Portion of the 4 Partition IRF to be Static

total IRF size include reducing energy consumption and decreasing access time.

We first read the instruction profile for the entire execution and assign the most frequently executed types of instructions to the static portion of the IRF. The functions are then partitioned among the IRF windows that have fewer entries using the algorithm described in Figure 7, where instructions in the static portion are not considered when assigning instructions to each window.

We varied the size of the static portion from 4, 8, 12, and 16 for the 4 window version of the IRF. Note that the basic IRF described in this paper always shares 1 static entry reserved for *nop*. Thus the static portion really only amounts to 3, 7, 11, or 15 shared meaningful instruction entries. The evaluated number of entries in the windowed portion of the IRF are 28, 24, 20, and 16, respectively. These values for the static portion correspond to a physical size reduction for the 4 window IRF of 10.4%, 20%, 29.6%, and 39.2%.

Figure 10 shows the results of holding a portion of the IRF entries static across the 4 hardware register windows. These results assume a uniform cost for accessing an IRF entry in each configuration, so increasing the static portion only reduces our ability to pack instructions. Fetch costs are increased approximately 0.07% by going to the 4 static entries, and moving to 16 static entries yields an average fetch cost of 53.97%. The actual hardware cost of moving to 4 partition with 16 shared entries is approximately 2.45 times greater than the cost of a single partition IRF, and the benefit is better than just adding a second window. A tuned implementation might obtain reductions in overall fetch energy consumption due to reduced leakage energy for the smaller physical size IRF, despite the reduced packing ability. Larger windowed IRFs may also be easier to support by reserving some registers statically for the entire application execution.

7 Using an IRF with a Loop Cache

We believe that using an IRF can be complementary, rather than competitive, with other architectural and/or compiler approaches for reducing energy consumption and compressing code size. One complementary technique for reducing instruction fetch energy is a loop cache. In this section we show that the combination of using an IRF and a loop cache reduces energy consumption much more than using either feature in isolation.

A loop cache is a small instruction buffer that is used to reduce instruction fetch energy when executing code in small loops [9]. We modified SimpleScalar to simulate the execution of the loop cache described by Lee, Moyer, and Arends [16]. This loop cache has three modes: *inactive*, *fill*, and *active*. The hardware detects when a short backward branch (*sbb*) instruction is taken whose target is within a distance where the instructions between the target and the *sbb* will fit into the loop cache. At that point the loop cache goes into *fill* mode, which means that the instructions are fetched from the IC and placed in the loop cache. After encountering the *sbb* instruction again, the loop cache is set to *active* mode, when instructions are only fetched from the loop cache. Whenever there is a taken transfer of control that is not the *sbb* instruction or the *sbb* instruction is not taken, the loop cache goes back to *inactive* mode and instructions are fetched from the IC. Thus, loop caches can be exploited without requiring the addition of any new instructions to the ISA.

One of the limiting factors for using this loop cache is the number of instructions within the loop. This factor can be mitigated to a large extent by the use of an IRF. For each tightly packed MISA instruction, there are up to five RISA instructions that are fetched from the IRF. The loop cache contains only the MISA instructions. Thus, the number of total instructions executed from the loop cache can potentially increase by a factor of five.

Figure 11 shows the instruction fetch cost of using a loop cache, a 4 window IRF, and a 4 window IRF with a loop cache normalized against using no IRF and no loop cache. We varied the size of the loop cache to contain 4, 8, 16, 32, and 64 instructions. The energy requirement of a loop cache access closely compares to that of a single IRF access. A MISA instruction fetched from the loop cache and that contains five RISA references would have an approximate fetch cost of 0.016 since there has to be a fetch from both the loop cache and the IRF before the first RISA instruction can be executed. We found that the loop cache and IRF can cooperate to further reduce instruction fetch costs. Whereas an 8-entry loop cache reduces the fetch cost to 93.26%, and a 4 window IRF can reduce fetch cost to 53.28%, the combination of such an IRF with a loop cache reduces the instruction fetch cost to 43.84%.

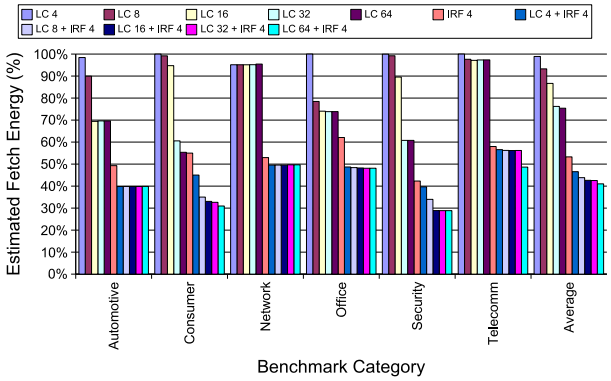

Figure 11. Reducing Fetch Energy with a Loop Cache and/or IRF

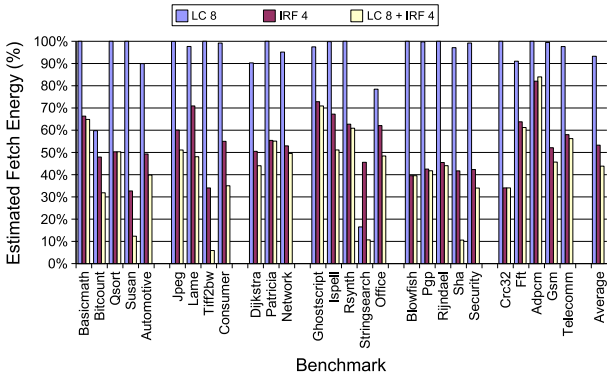

Figure 12. Reducing Fetch Energy Using a Combination of 8-entry Loop Cache and 4 Partition IRF

The results show clearly that a loop cache and an IRF can operate in a complementary fashion, often obtaining better results than either can separately. One limitation of a loop cache is that the maximum loop size is fixed, and any loop that extends beyond the *sbb* distance can never be matched. Packing instructions naturally shortens this distance for frequently executed instructions, thus allowing more loops to fit into the loop cache. The loop cache is also limited to loops that do not contain call instructions as any taken transfer of control invalidates the cache contents. Lengthening the actual size of the loop cache only provides greater access for loops with transfers of control (e.g. outer loop nests and those with function calls) to start the filling process. Our IRF does not currently support packing call instructions (since they are rare and usually diverse), and thus these references remain a single unpacked MISA instruction. Additionally, packed instructions can contain at most one branch instruction, which terminates the pack execution. Due to these factors, code packed for IRF tends to be less dense around sections containing potential transfers of control, leading to longer branch distances for the *sbb* and serves in essence as a filter for eliminating these poor loop choices. Instruction packing with an IRF is limited in the number of RISA instructions available from a single MISA instruction that is fetched from memory. Thus, there is an approximate lower bound of 21.5% for fetching with an IRF as at least 1 out of every 5 instructions (assuming all tight packs) must be MISA. The loop cache can assist by providing low-power access to frequently executed packed MISA instructions, thus allowing the fetch cost with an IRF to drop even further. In this manner, both schemes are able to alleviate some of their inefficiencies by exploiting the features provided by the other.

Figure 12 provides a more detailed look at the interactions between an 8-entry loop cache and a 4 window IRF. Notice that for Susan, Tiff2bw, Stringsearch, and Sha, the fetch cost is reduced below 15% (saving more than 85%! of the fetch energy over the baseline model) for the combined IRF and loop cache scheme. These benchmarks demonstrate the potential multiplicative collaboration that exists between a loop cache and an IRF performing at their peak.

8 Crosscutting Issues

There are a couple of design issues that must be addressed to incorporate an IRF into a processor architecture. Using IRF register windows means more state has to be preserved at context switches. The IRF need not be saved at each context switch since the instructions in each window are unchanged copies of a section of memory. However, the IRF may need to be reloaded after returning from a context switch. Rather than eagerly reloading all of the IRF windows, the processor could only reload a particular window when it is referenced and is not resident in the IRF. An exception could be generated when such a situation is encountered, which should only infrequently occur. We are exploring an alternate caching model (discussed in the next section) that would moderate the impact of saving IRF state on a context switch.

Another issue involves library code compiled separately from application code. A number of solutions are possible in this case. First, the library code could avoid using packed instructions – though this would limit the advantage of the IRF. Second, a software convention could specify a single IRF window containing common instructions for a wide variety of library routines that is always mapped to a specific IRF window. Any call to a library routine would specify the generic library IRF window. This solution requires identification of library routines so the compiler can

use the correct IRF window. Finally, IRF loads can be used to overwrite IRF entries within the library routines as long as the IRF state can be restored when the application code is reentered. Note that we do not attempt to pack instructions in library functions into the IRF and do not include the results of executing library code in our measurements.

9 Future Work

There are many areas of future work that are possible to investigate for dynamically selecting the instructions to place in the IRF. In this paper, we explored using IRF load instructions to change the contents of the IRF and we examined the benefits of multiple IRF windows. While we carefully chose the number of IRF windows to match the average number of different partitions chosen by the compiler, there are some applications that would benefit from having additional windows. There are two ways to achieve a larger set of windows than there are hardware windows: The first is to return to the IRF load instructions used in Section 4, but with the multiple IRF hardware windows proposed in Section 5. For most applications this would not provide additional performance gains (since 4 IRF windows is sufficient), but for some of the larger applications, this would further improve our ability to pack more instructions. Furthermore, it would come at much lower overhead costs, since IRF loads only need occur when the program switches to a phase that cannot be best serviced by one of four resident IRF windows (instead of the only resident IRF in the results in Section 4). Alternately, the microarchitecture can virtualize a large set of windows (say 32), and rely on caching to retain the most active subset in the physical IRF. This would be organized as a fully associative set of 4 cached IRFs supporting a memory resident set of IRF windows. In this configuration, the call instruction still specifies which of the 32 windows to switch to; if that window is not resident in the 4-entry IRF cache, a miss occurs and the LRU entry is replaced with the IRF window specified in the call.

Another area of future research is to expand the evaluation of the IRF with existing code compression techniques. Many of the existing code compression techniques require a multiple cycle fetch in order to perform the decompression. In a conventional instruction fetch pipeline, there is little opportunity to hide the increased latency of the multiple cycle access, but instruction fetches from memory occur less frequently with an IRF as it takes multiple cycles to consume packed instructions. That is, a single instruction fetched from the IC may specify up to 5 instruction to fetch from the IRF. Most processor pipeline implementations, especially low-power embedded processors, could not process 5 instructions in a single cycle. However, the next IC fetch can occur immediately after (even in the event of a branch in the IRF), so when the instructions fetched from the IC contain a packed instruction, the following IC fetch can often take an additional cycle without incurring any performance penalty. This synergistic relationship between the IRF and code compression can result in a significant reduction in any performance degradation expected with the decompression algorithm. Alternately, it can enable more sophisticated compression algorithms by subsuming much of the latency effects.

10 Conclusions

In this paper, we have examined several extensions to incorporating an IRF into a processor architecture. These include software inserted by the compiler to modify the contents of the IRF as program phase changes occur. This approach improves the ability of an IRF to reduce energy requirements for instruction fetch — particularly for those large applications that exhibit many diverse execution phases. We also propose a new windowed IRF mechanism that further reduces instruction fetch energy by eliminating almost all of the overhead from switching IRF windows between functions. Our results show that the use of IRF windows can eliminate almost $1/2$ of the instruction fetch energy requirements on average across a wide selection of embedded applications. We also showed the synergistic relationship between the use of an IRF and existing loop cache designs to further reduce energy requirements — to less than 10% of the baseline instruction fetch energy for some applications. As with the earlier results on employing an IRF, our research showed that the addition of an IRF can significantly reduce fetch energy without negative tradeoffs in code size (reduced with IRF) and execution time (slightly reduced due to IC effects).

The IRF can be easily integrated into an embedded processor pipeline architecture and can be effectively utilized by compilers using profiled feedback or compile-time estimates of instruction frequencies. IRF support can be easily added to an existing architecture, requiring only a few free opcodes to implement, and since the RISA architecture can differ from the MISA, an IRF can provide a simple mechanism for extending an ISA with few remaining instruction encoding bits.

11 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] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35:59–67, February 2002.

[2] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 329–338. ACM Press, 1988.

[3] A. Cheng, G. Tyson, and T. Mudge. FITS: Framework-based instruction-set tuning synthesis for embedded application specific processors. In *DAC '04: Proceedings of the 41st annual conference on Design Automation*, pages 920–923, New York, NY, USA, 2004. ACM Press.

[4] K. Cooper and N. McIntosh. Enhanced code compression for embedded risc processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 139–149, May 1999.

[5] M. Corliss, E. Lewis, and A. Roth. A DISE implementation of dynamic code decompression. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 232–243, June 2003.

[6] S. K. Debray, W. Evans, R. Muth, and B. DeSutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[7] J. Eyre and J. Bier. DSP processors hit the mainstream. *IEEE Computer*, 31(8):51–59, August 1998.

[8] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 117–121, June 1984.

[9] A. Gordon-Ross, S. Cotterell, and F. Vahid. Tiny instruction caches for low power embedded systems. *Trans. on Embedded Computing Sys.*, 2(4):449–481, 2003.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[11] S. Hines, J. Green, G. Tyson, and D. Whalley. Improving program efficiency by packing instructions into registers. In *Proceedings of the 2005 ACM/IEEE International Symposium on Computer Architecture*, pages 260–271. IEEE Computer Society, 2005.

[12] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. 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*, pages 219–230, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[13] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proceedings of the 1997 International Symposium on Microarchitecture*, pages 184–193, 1997.

[14] K. D. Kissell. MIPS16: High-density MIPS for the embedded market. Technical report, Silicon Graphics MIPS Group, 1997.

[15] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder. Reducing code size with echo instructions. In *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 84–94. ACM Press, 2003.

[16] L. Lee, B. Moyer, and J. Arends. 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*, pages 267–269, 1999.

[17] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proceedings of the 1997 International Symposium on Microarchitecture*, pages 194–203, December 1997.

[18] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160-mhz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Tech. J.*, 9(1):49–62, 1997.

[19] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, Thumb, and the ARM7TDMI. *IEEE Micro*, 15(5):22–30, October 1995.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM Press.

[21] SimpleScalar-ARM Power Modeling Project. http://www.eecs.umich.edu/∼panalyzer.

[22] M. Smith, M. Horowitz, and M. Lam. Efficient superscalar performance through boosting. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, October 1992.

[23] D. Weaver and T. Germond. *The SPARC Architecture Manual*, 1994.

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