Improving Program Efficiency by Packing Instructions into Registers

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

Abstract

New processors, both embedded and general purpose, often have conflicting design requirements involving space, power, and performance. Architectural features and compiler optimizations often target one or more design goals at the expense of the others. This paper presents a novel architectural and compiler approach to simultaneously reduce power requirements, decrease code size, and improve performance by integrating an instruction register file (IRF) into the architecture. Frequently occurring instructions are placed in the IRF. Multiple entries in the IRF can be referenced by a single packed instruction in ROM or L1 instruction cache. Unlike conventional code compression, our approach allows the frequent instructions to be referenced in arbitrary combinations. The experimental results show significant improvements in space and power, as well as some improvement in execution time when using only 32 entries. These advantages make packing instructions into registers an effective approach for improving overall efficiency.

1. Introduction

Embedded systems are subject to a variety of design constraints. Performance must be sufficient to meet the timing constraints for the application. Power consumption should be minimized, often to be less than a specified target. The size of read-only memory (ROM) may have rigid limits to minimize cost. Unfortunately, it is often difficult to improve one parameter without negatively affecting others; increasing clock frequency to enhance performance also increases power consumption; code compression techniques improve code density, and voltage scaling reduces power requirements, but these may increase execution time. While this has been the prevailing design environment for embedded processors, these same design constraints now challenge general-purpose processor design as well. To address each design issue (power, code size, and execution time), we must identify inefficiencies in some part of program execution that can be improved by devising new micro-architectural resources, instruction set design improvements, and/or compiler optimizations. In this paper, we use all three techniques to improve instruction fetch (I-Fetch) logic.

Optimizing I-Fetch logic is a natural target for embedded processors: it consumes approximately 36% of total processor power on a StrongARM [18]. Current I-Fetch mechanisms are inefficient in at least two aspects. First, information content of instruction encoding techniques is far from theoretical compression limits: instruction encoding seeks to maximize functionality of the ISA while simplifying decoding by usually making all instructions the same length, even though most applications use only a fraction of available instruction encodings. Second, inefficiency in I-Fetch involves the instruction storage structure: instruction cache (IC) or instruction ROM. These structures use less power and have lower latency than a large memory store, but they allocate a large, flat storage method on all instruction accesses. This approach requires almost all references to be accessed from the same storage (IC hit or ROM access) with the same power costs, even though only a small subset of instructions account for the majority of references.

The analogous problem with data references has been managed through the use of data registers via compile-time control of register allocation. We propose extending the use of registers to store frequently referenced instructions. This enables the compiler to place those instructions referenced most frequently into a small register file that can be indexed with a small register specifier. The small size of the register file reduces power requirements for frequent instruction references and use of an instruction register specifier reduces code size.

Two new SRAM structures are required to support our I-Fetch optimization. The first is an instruction register file (IRF) to hold common instructions as specified by the compiler. The second is an immediate table (IMM) containing the most common immediate values used by instructions in

0-7695-2270-X/05/$20.00 (C) 2005 IEEE

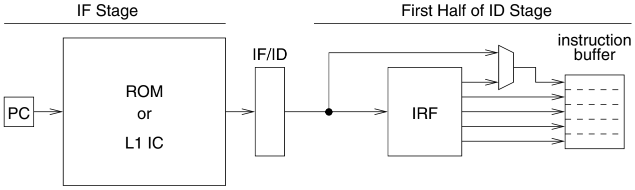


Figure 1. Instruction Register File

the IRF. We modify the ISA to support fetching multiple instructions from the IRF. By specifying multiple IRF instructions within a single 32-bit instruction fetched from the IC, we can reduce the number of instructions in the program (saving space), reduce references to the IC (saving power), and effectively fetch more instructions per cycle (improving execution time).

The IRF can be integrated into the pipeline architecture in one of two locations — at the end of instruction fetch or the start of instruction decode. Our Verilog models for simple pipelines show that there is sufficient time in either stage to perform the IRF access, since the critical path for our designs is in the execute stage. If this is not the case for some other pipeline, then the IRF can be modified to hold partially decoded instructions, and IRF access can be overlapped from the end of instruction fetch to the start of instruction decode, reducing cycle time impact. Figure 1 shows the integration of the IRF at the beginning of instruction decode. If the instruction fetched from the IC is a packed instruction, instruction index fields select from one to five IRF entries to write to the instruction buffer. The buffer is also modified to accommodate multiple instruction insertion. Not shown in Figure 1 is the the IMM, which sits next to the IRF and contains 32-bit immediate values. These can be associated with one or more IRF instructions through the use of *parameterized* packed instructions (see Section 2). In the event that an IRF instruction uses an immediate from the IMM, the value is concatenated to the instruction bits from the IRF. By parameterizing use of immediates and placing them in a separate register file, we can support full 32-bit immediate values and can place two or more instructions that differ only by immediate value into the same IRF entry.

Figure 2 illustrates the potential for storing instructions in an IRF. We select the largest application from each of the six categories of the MiBench suite, profiling each with its small input set, and gathering dynamic instruction counts for each distinct instruction executed [12]. The data is sorted (high to low) into a cumulative plot. The x-axis shows sorted distinct instructions, and the y-axis shows the percentage of dynamic instruction fetches to the most common instructions. The graph shows the percentage of instruction fetches that can potentially be captured with an IRF of various sizes. On average, about 66.51% of all instructions executed can be stored in a 32-entry IRF, assuming it can be loaded with the 32 most common instructions at the start of execution.

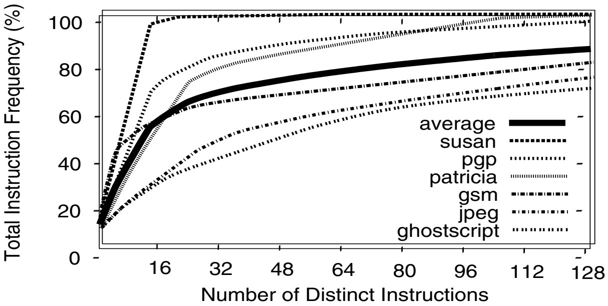


Figure 2. Dynamic Instruction Redundancy

The remainder of this paper is organized in the following manner. First, we describe the architectural extensions necessary for supporting an instruction register file. Second, we outline the modifications made to support compiling for an architecture that uses instruction registers. Third, we analyze experimental results showing the effect of packing instructions into registers on code size, power, and execution time. Fourth, we present some crosscutting issues regarding the support costs of making an IRF available in an ISA. Fifth, we review a variety of past compiler and architectural solutions that attempt to save space by compressing programs and/or energy by reducing the fetch cost of instructions. Sixth, we mention several potential topics for future work. Finally, we present our conclusions regarding instruction packing and the IRF.

2. ISA Modifications

This section describes the changes necessary for an ISA to support references to a 32-instruction register file. We choose the MIPS ISA, since it is commonly known and has a simple encoding [20]. Instructions stored in memory will be referred to as the *Memory ISA* or *MISA*. Instructions we place in registers will be referred to as the *Register ISA* or *RISA*. Note that the MISA and RISA need not use exactly the same instruction formats, however this paper presents only minor modifications to the RISA to reduce complexity. MISA instructions that reference the RISA are designated as being *packed*.

2.1. Loosely Packed Instructions

The simplest type of packed instruction is the *loosely packed instruction*. An instruction using this format allows two instructions to be fetched for the price of one by improving use of encoding space in the traditional MIPS instruction formats. In a loosely packed instruction, a standard MIPS instruction is modified to contain an additional reference to an instruction from the IRF. The standard in-

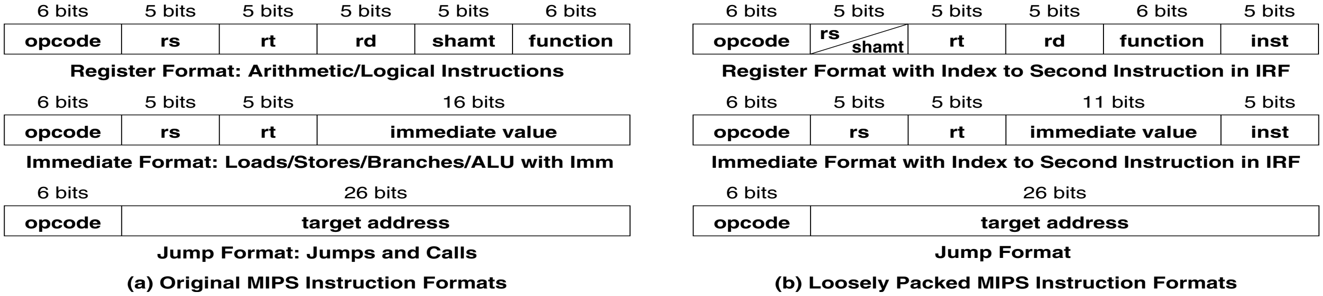

6 bits
5 bits
5 bits
5 bits
5 bits
6 bits
opcode
rs
rt
rd
shamt
function
Register Format: Arithmetic/Logical Instructions
6 bits
5 bits
5 bits
16 bits
opcode
rs
rt
immediate value
Immediate Format: Loads/Stores/Branches/ALU with Imm
6 bits
26 bits
opcode
target address
Jump Format: Jumps and Calls
(a) Original MIPS Instruction Formats
6 bits
5 bits
5 bits
5 bits
6 bits
5 bits
opcode
rs
shamt
rt
rd
function
inst
Register Format with Index to Second Instruction in IRF
6 bits
5 bits
5 bits
11 bits
5 bits
opcode
rs
rt
immediate value
inst
Immediate Format with Index to Second Instruction in IRF
6 bits
26 bits
opcode
target address
Jump Format
(b) Loosely Packed MIPS Instruction Formats

Figure 3. MIPS Instruction Format Modifications

struction is executed followed by the IRF instruction. One IRF entry is reserved for a *nop*, to indicate that no additional instruction is executed when a standard instruction is followed by an instruction not in the IRF.

Figure 3(a) shows traditional MIPS instruction formats, and Figure 3(b) shows the proposed modifications to allow loose packing. The IRF *inst* field replaces the infrequently used *shamt* field of an R-type instruction. Similarly, for the I-type instruction format, we reduce the size of the *imm* field to include an IRF instruction reference. J-type instructions are left unchanged.

Several restrictions must be placed on the MIPS ISA to support these adaptations. Immediate values are reduced from 16 bits to 11 bits. We must therefore change the *lui* instruction, which loads a 16-bit value into the upper half of a register and is used to construct large constants or addresses. The modified *lui* loads upper immediate values of 21 bits by using the standard 16 immediate bits of an I-type instruction along with the previously unused 5-bit *rs* field. The new *lui* instruction cannot be loosely packed with an additional instruction since all 32 bits are used. Removing the *shamt* field from R-type instructions forces shift instructions to change format as well. Shifts are now encoded with the *shamt* value replacing the *rs* register value, which is unused in the original format when shifting by an immediate. Additional opcodes or function codes to support the loosely packed format are unnecessary.

Although the IRF focuses on packing common instructions together into large packs, the loosely packed instruction serves an important role. Often an individual IRF instruction can be detected, yet neither of its neighboring instructions are available via the IRF. Without loosely packing instructions, we could not capture this type of redundancy, and thus we could not achieve the same levels of improvement.

2.2. Tightly Packed Instructions

Figure 4 shows the added *T-type* format for tightly packed instructions. This format allows several RISA instructions to be simultaneously accessed from a single MISA instruction. Since the IRF contains 32 entries, a

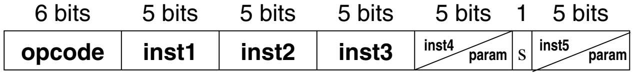


Figure 4. Tightly Packed Format

tightly packed instruction can consist of up to five instructions from the IRF. Tightly packed instructions can support fewer than five RISA instructions by padding with *nop* references. Hardware support halts execution of the packed instruction when a *nop* is encountered (so there is no performance degradation).

2.3. Using Parameterized Immediate Values

Preliminary studies show that I-type instructions account for 51.91% of static instructions and 43.45% of dynamic instructions executed. Further examination of immediate values present in these instructions reveals that many common values are used in a variety of instructions. By parameterizing the IRF entries that refer to immediate values, more instructions can be matched for potential packing. Parameterization however requires additional bits to encode the necessary value to fetch. The T-type instruction format, shown in Figure 4, provides encoding space for up to two immediate parameter values for each tightly packed instruction.

We find that using a small range of values is much less effective than referencing the most frequently used immediates. For example, when five bits are used, the static percentage of immediates represented increases from 32.68% to 74.09%, and the dynamic percentage increases from 74.04% to 94.23%. Thus, we keep the immediate values in an immediate table (IMM), so that a packed instruction can refer to any parameter value. Each IRF entry also contains a default immediate value, so that these instructions can remain loosely packable or can be referenced with no parameters in a tightly packed instruction. The IMM contains 32 entries, so five bits are used to access each parameter.

There are two major options available for parameterized instruction packs. One option consists of up to four IRF entries along with a single parameter. The other option allows for up to three IRF entries with two parameters. The opcode used with the encoding of the *S* bit dictates which IRF in-

#	RTL	RTL (positional)
1	r[2]=R[r[29]+4];	r[2]=R[r[29]+4];
2	r[2]=r[2]+r[5];	**s[0]**=**s[0]**+r[5];
3	R[r[29]+4]=r[2];	R[**u[2]**+4]=**s[0]**;
	...	...
4	r[3]=R[r[29]+4];	r[3]=R[r[29]+4];
5	r[3]=r[3]+r[5];	**s[0]**=**s[0]**+r[5];
6	R[r[29]+4]=r[3];	R[**u[2]**+4]=**s[0]**;

Figure 5. Positional Register Specifiers

struction uses each available parameter, while other I-type instructions in the pack use their default immediate values.

The T-type instruction format requires additional opcodes to support tightly packed instructions. There are four different instructions that can use the single parameter when referencing an instruction containing up to four IRF entries. Similarly, there are three possibilities for using two parameters when referencing an instruction containing up to three IRF entries. Including the tightly packed instruction with no parameterization, there are eight new operations to support. Fortunately the *S* bit allows these eight operations to be represented with four opcodes.

2.4. Using Positional Registers

Another method for increasing instruction redundancy is to detect and exploit common patterns in register usage. Register accesses may be referred to positionally, such that a register previously set or used can be referenced by the current instruction. In this way, a RISA instruction has greater flexibility than a MISA instruction, since it can access both physical hardware registers as well as the newly added positional register specifiers. Positional registers that are set and used can be tracked via the pipeline registers already available in hardware. Positional specifier register fields are hereafter referred to as *prs*, *prt*, and *prd*, corresponding to the standard MIPS fields *rs*, *rt*, and *rd* [20].

Figure 5 illustrates conversion to positional register form using two distinct sequences of load/add/store operations. In line 2, the reference to r[2] can be abstracted out as the previous instruction's set register. Since r[5] has not been referenced in a nearby previous instruction, it cannot be converted to a positional parameter. Line 3 shows that again we can refer to the prior set as s[0] and the use of r[29] as u[2]. At most one set and two uses can be present in any standard MIPS instruction, so u[2] refers to the first use (*rs*) of the instruction that is located two instructions back (three uses back). Lines 5-6 match lines 2-3 when using the positional register specifiers, even though they reference different architectural registers. This enables lines 2 and 5 as well as lines 3 and 6 to share the same IRF entries. Only RISA instructions can access registers positionally. This architectural enhancement exposes greater instruction redundancy, and thus can lead to higher density instruction packing.

2.5. Packing Branches and Jumps

Branch instructions are troublesome to pack directly since the packing process will undoubtedly change many branch offset distances. The addition of parameterized instruction packs greatly simplifies packing of branches. Preliminary testing shows that approximately 63.15% of static branches and 39.75% of dynamic branches can be represented with a 5-bit displacement. Thus, a parameter linked to a RISA branch instruction would refer to the actual branch displacement and not to the corresponding entry in the IMM. Once packing is completed, branch displacements can be recalculated and inserted into the corresponding packed instructions.

MIPS jump instructions use the J-type format, and are not easily packed due to the use of a target address. To remedy this, we encode an unconditional jump as a conditional branch that compares a register to itself for equality when the jump offset distance can be represented in five bits. This allows IRF support of parameterizable jump entries.

3. Compiler Modifications

This section provides an overview of the compilation framework and modifications to carry out the experiments we describe. The compiler is a port of VPO (Very Portable Optimizer) for the MIPS architecture [3]. Additional modifications allow it to be used with the PISA target of the SimpleScalar toolset for gathering performance statistics [2].

The hardware modifications listed in Section 2 have a definitive impact on the code generated by the compiler. Immediate values are constrained to fit within 11 bits, or are constructed in similar fashion as traditional immediates with sizes greater than 16 bits. We find that this can slightly lengthen code for an application, but initial size increases are far overshadowed by the benefit of instruction packing.

The GNU assembler MIPS/PISA target provided for the SimpleScalar toolset allows the use of a variety of pseudoinstructions to ease the job of the compiler writer. These pseudoinstructions are later expanded by the assembler into sequences of actual hardware instructions. Detection of RISA instructions at compile time is unnecessarily complicated by the presence of such pseudoinstructions. To reduce this complexity, we expand most of these pseudoinstructions at compile time, prior to packing instructions. The load address pseudoinstruction is not expanded until assemble/link-time since we currently do not pack instructions that refer to global memory addresses. A preliminary study for the MIPS/PISA target shows that even packing all load address instructions yields very little improvement due to their relative infrequency.

We modified VPO and SimpleScalar to incorporate additional information to facilitate instruction profiling for

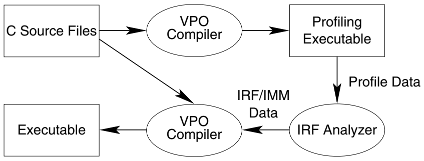


Figure 6. Compiling for IRF

placement within the IRF. One important aspect is specification of instruction operands using positional registers for prior register sets and uses. This allows the profile to contain both standard and positional register specifiers for each particular instruction, whether selection is done for code compaction or maximizing instances of fetching out of the IRF.

Figure 6 shows the flow of information at compile-time. VPO and the PISA toolchain are used to create the executable, which can then be executed and/or profiled using SimpleScalar. Profile information is then extracted and passed to an analysis tool for IRF instruction selection. This tool then supplies the compiler with the new IRF and Immediate Table entries to use when recompiling the application. Note that it is possible to remove the need for profiling by allowing the compiler to approximate dynamic frequencies by looking at loop nesting depths and the entire program control flow graph.

3.1. Selecting IRF-Resident Instructions

The IRF is clearly a limited storage space, so instructions selected to populate it must provide the greatest benefit to program performance. Calculating the optimal set of instructions would be too time-consuming during compilation, so a heuristic is used. The current method uses a greedy algorithm based on profiling the application and selecting the most frequently occurring instructions as well as default immediate values.

Figure 7 shows the process of selecting the IRF-resident (RISA) instructions. The algorithm begins by reading the instruction profile generated by either static and/or dynamic analysis. Next, the top 32 immediate values are calculated and inserted as part of the IRF for parameterized immediate values. Each I-type instruction that uses one of the 32 immediate values is now a candidate for parameterization, and it is combined with I-type instructions that are identical except for referring to different parameterizable immediate values. The most frequent immediate value from each I-type group is retained as the default value.

At this point, the instructions are placed in two lists. One list contains positional form instructions that refer to prior sets and uses of registers within a few instructions back. The other list contains physical hardware register references. When constructing these lists, careful attention must be paid to operand specifiers, since elements should be grouped together if they have the same operand fields. In the physical register instruction list, positional specifiers can be safely ignored. As for the positional form instruction list, any specified positional register in one operand replaces the corresponding physical register reference, so *prs* replaces *rs*, *prt* replaces *rt*, and *prd* replaces *rd*. Comparisons are then made on these fields to group instructions appropriately.

```
Read in instruction profile (static or dynamic);
Calculate the top 32 immediate values for I-type instructions;
Coalesce all I-type instructions that match based on
   parameterized immediates;
Construct positional and regular form lists from the
   instruction profile, along with conflict information;
IRF[0] ← $nop$;
foreach $i \in [1..31]$ do
    Sort both lists by instruction frequency;
    IRF[i] ← highest freq instruction remaining in the two lists;
    foreach conflict of IRF[i] do
        Decrease the conflict instruction frequencies by the
           specified amounts;
```

Figure 7. Selecting IRF Instructions

Opcode	rs	rt	immed	prs	prt	Freq
addiu	r[3]	r[5]	**1**	s[0]	NA	400
addiu	r[3]	r[5]	**4**	s[0]	NA	300
addiu	r[7]	r[5]	1	s[0]	NA	200
...						

⇓ Coalescing Immediate Values ⇓

addiu	**r[3]**	r[5]	1	s[0]	NA	700
addiu	**r[7]**	r[5]	1	s[0]	NA	200
...						

⇓ Grouping by Positional Form ⇓

addiu	NA	r[5]	1	s[0]	NA	900
...						

⇓ Actual RTL ⇓

r[5]=s[0]+1	900

Figure 8. Coalescing Similar Instructions

After constructing the lists, a *nop* is added as the first IRF element. This guarantees that one instruction followed by another not in the IRF can be handled without wasting processor cycles. At this point the algorithm enters its main loop to select the other 31 IRF entries. Each list is sorted by instruction frequency. The initial sort needs to be complete; later sorts need only move elements with modified frequencies due to conflicts. The highest frequency instruction is selected for the IRF and removed from its corresponding list. Conflicts for this instruction from the other list are examined, and their frequency counts are adjusted.

Figure 8 shows coalescing of immediates as well as grouping of positional parameters. In this example, the immediate values 1 and 4 are considered to be available as parameters to the IRF. This allows the first two *addiu* instructions to be grouped. The higher frequency immediate (1) is retained as the default value in the second part of the figure. Next, the figure shows grouping by positional form for these elements. In this case, the *prs* field supersedes the *rs* field, allowing both instructions to match. The final output RTL for the IRF entry is depicted at the bottom, along with its updated final frequency count of 900.

Commutative operators may yield different operand orderings for the same semantic operations. This inability to recognize that two different instructions are semantically equivalent can inhibit selection of IRF candidates and thus lower overall effectiveness of instruction packing. For this reason, we add a transformation to VPO that converts all commutative operations into a single form. We reorder register operands such that if one source operand matches the result register, this operand is placed first. If neither operand matches the result register, we choose the register with lowest number to be placed first. Figure 9(a) shows swapping operands r[3] and r[4], since the result is stored in register r[4]. In Figure 9(b), registers r[3] and r[4] are reordered by register number.

	Before	After
(a)	`r[4]=r[3]&r[4];`	`r[4]=`**`r[4]`**`&r[3];`
(b)	`r[2]=r[4]|r[3];`	`r[2]=`**`r[3]`**`|r[4];`
(c)	`r[5]=r[3];`	`r[5]=r[3]`**`+0`**`;`
(d)	`r[7]=95;`	`r[7]=`**`r[0]+`**`95;`

Figure 9. Equivalent Instructions

Besides commutativity, the effects of a particular instruction may be representable by a different instruction. One common case is the *mov* pseudoinstruction, which is by default converted into an *or* or an *add* instruction with r[0] (hardwired on the MIPS to have a zero value). To increase our ability to pack instructions, we instead choose to map these instructions as *addi* instructions with default parameter zero, providing maximum flexibility for matching with parameterization later. This effectively allows us to capture all *mov* instructions while potentially recognizing some less common increment instructions for free. Figure 9(c) shows a register *mov* operation being converted to an *addi* with zero instead of an *add* with r[0]. Figure 9(d) shows an *li* similarly converted to an *addi* with r[0].

3.2. Packing Instructions

Since packing is done by the compiler, the source program must be recompiled after IRF selection. After profiling the original optimized application, VPO is supplied with both an IRF as well as the top immediate values available for parameterization via the IMM. Each optimized instruction is examined first for direct matches, and then tested for matching the IRF with a parameterized immediate value. The instruction is marked as either present in the IRF and/or present in the IRF with parameterization, or not present. Once each instruction is categorized, the compiler proceeds with packing the appropriately marked instructions.

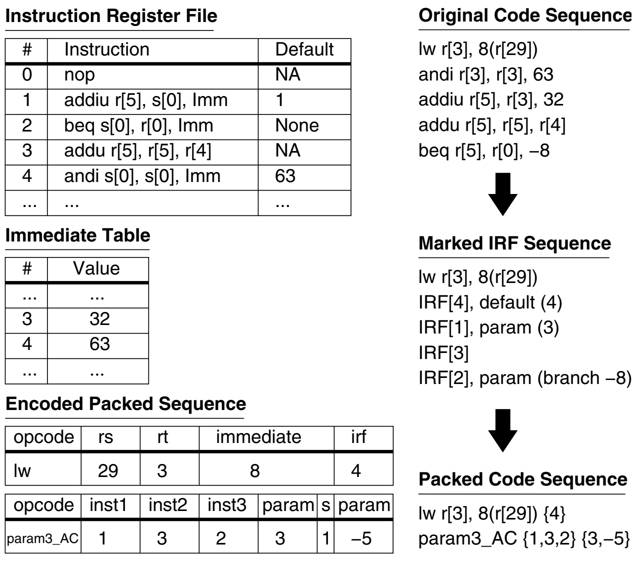
Instruction packing is performed per basic block, and we currently do not allow instructions to be packed across basic block boundaries. Targets of control transfers must be the address of an instruction in memory (not in the middle of a tightly packed instruction). Table 1 summarizes the different packed instruction types that are currently available in decreasing order of preference. The packing algorithm operates by examining a sliding window of instructions in each basic block in reverse order. The algorithm attempts each of the pack types in turn, until it finds a match. Packs made up of instructions only are preferred over parameterized packs referencing the same number of instructions. The tightly packed format supports up to five IRF entries, with the unused slots occupied by the *nop* located at IRF[0]. When a pack is formed, the instructions are merged into the appropriate instruction format. The sliding window is then moved so that the next set of instructions in the block can be packed.

Table 1. Packed Instruction Types

Name	Description
tight5	5 IRF instructions (no parameters)
tight4	4 IRF instructions (no parameters)
param4	4 IRF instructions (1 parameter)
tight3	3 IRF instructions (no parameters)
param3	3 IRF instructions (1 or 2 parameters)
tight2	2 IRF instructions (no parameters)
param2	2 IRF instructions (1 or 2 parameters)
loose	Loosely packed format
none	Not packed (or loose with nop)

After instruction packing is performed on all basic blocks, packing is re-attempted for each block containing a branch or jump that does not reside in a tightly packed instruction. One insight that makes iterative packing attractive is the realization that branch distances can decrease due to instruction packing. This decrease can cause branches or jumps that were previously not parameterizable via the IRF to slip into the 5-bit target distance (-16 to +15). After detecting this, packing is then re-applied to the basic block. If the end result is fewer instructions, then the new packing is kept; otherwise the branch is ignored, and the block is restored to its prior instruction layout. Any changes that cause instructions to be packed more densely triggers re-analysis of IRF instructions, and the process continues until no further changes are made.

Figure 10 shows instruction packing for a simple sequence of operations. The IRF shows five entries including *nop*, and the IMM includes 32 and 63 as possible values. The original code is mapped to corresponding IRF entries, both with and without parameterization. The branch can be parameterized since -8 can be encoded as a branch offset within the 5-bit parameter field. Examination of the basic block reveals four instructions present in the IRF. However this sequence requires two immediate parameterizations, which is only possible when packing groups of three instructions or fewer. Thus, the code sequence is packed by grouping the last three instructions together as a *param3_AC* instruction. The *AC* denotes the instructions receiving the parameters, in this case the first and third. Moving the slid-



Instruction Register File

#	Instruction	Default
0	nop	NA
1	addiu r[5], s[0], Imm	1
2	beq s[0], r[0], Imm	None
3	addu r[5], r[5], r[4]	NA
4	andi s[0], s[0], Imm	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

Encoded Packed Sequence

opcode	rs	rt	immediate	irf
lw	29	3	8	4

opcode	inst1	inst2	inst3	param	s	param
param3_AC	1	3	2	3	1	–5

Original Code Sequence

```
lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8
```

Marked IRF Sequence

```
lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)
```

Packed Code Sequence

```
lw r[3], 8(r[29]) {4}
param3_AC {1,3,2} {3,–5}
```

Figure 10. Packing Instructions with an IRF

Table 2. MiBench Benchmarks

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

ing window past the last three instructions, we see a default parameter IRF instruction along with a non-IRF instruction. These two are combined into a single loosely packed instruction. The branch offset is adjusted to -5, since three preceding instructions have been compressed via the IRF. The breakdown of the various fields in each of the packed instructions is also shown in the figure.

4. Results

To measure the efficacy of packing instructions into registers, we select several benchmarks from the MiBench suite. MiBench consists of six categories of embedded software applications, each containing multiple entries. Table 2 shows the benchmarks used to determine the potential benefits of using an IRF with respect to code size, energy consumption, and execution time.

An instrumented version of SimpleScalar version 3.0 is used to collect relevant data for each benchmark [2]. For the purpose of this study, only the source code provided for each benchmark was subjected to profiling and instruction packing. Library code, although linked statically for SimpleScalar, is left unmodified.

Figure 11 shows the average static instruction redundancy. This serves as a limit for packing instructions into an IRF. Three curves are shown, corresponding to distinct instructions, instructions with parameterized immediates, and instructions with parameterized immediates and positional register specifiers. Although positional register specifiers do not appear to provide much additional benefit over instructions with parameterized immediates, it is important to realize that actual instruction packing can be limited by other factors such as basic block boundaries and instructions that cannot be loosely packed such as *lui*.

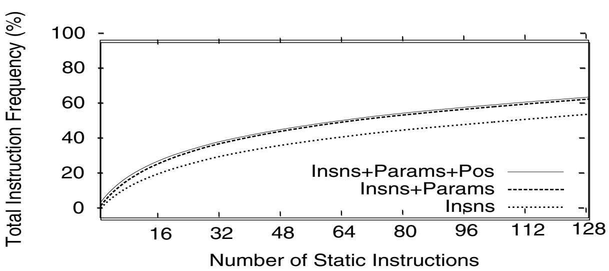


Figure 11. Static Instruction Redundancy

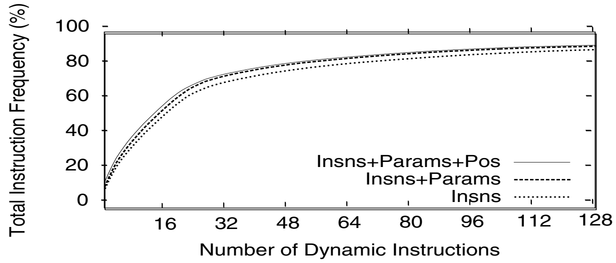


Figure 12. Dynamic Instruction Redundancy

The average dynamic instruction redundancy over all benchmarks is shown in Figure 12. This curve is visibly steeper than the static instruction redundancy curve, since applications execute instructions from critical loops with a much greater frequency. Additionally, not every instruction will be executed for a given set of test data, so the 100% mark is reachable with fewer of these frequently occurring instructions.

Table 3 compares the instruction mix by IRF type when profiling for static code compression versus dynamic execution. The percentages are calculated using fetches from the IC. Thus fetching an unpacked instruction and a packed instruction are equivalent, but the packed instruction represents more executed instructions. With dynamic profiling, we see larger pack sizes, since we can pack many instructions from the same frequently-executed loop bodies. Static packing can pack more instructions overall, but relies heavily on the loosely packed format.

4.1. Static Code Size

Figure 13 shows the reduction in code size for instruction packing with an IRF, as well as the enhancements made. 100% corresponds to the initial size of the compiled code without packing. Packing instructions alone reduces code to

Table 3. Instruction Mix with 32-entry IRF

Pack Type	Static %	Dynamic %
Not packed	84.94	65.24
Loosely packed	3.91	3.40
Tight2	3.22	3.19
Tight3	0.90	2.60
Tight4	0.98	3.14
Tight5	1.49	13.35
Param2	2.07	2.85
Param3	1.38	1.61
Param4	1.09	4.60

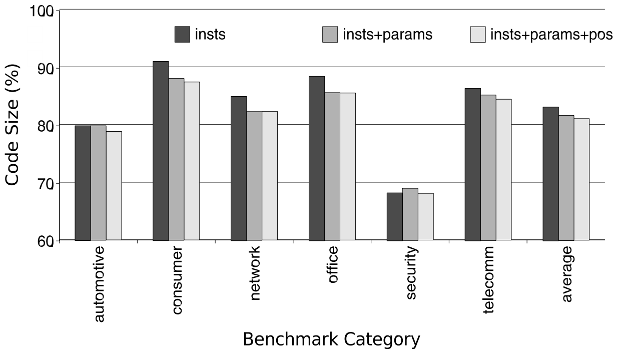


Figure 13. Reducing Static Code Size

83.23% of its original size. Parameterizing immediate values with the help of the IMM further reduces code size to 81.70%. Incorporating positional register specifiers reduces code to 81.09% of its original size, on average.

Clearly, parameterizing immediates increases our ability to pack instructions. For instance, we find that function prologue and epilogue code on the MIPS is easily parameterized. This leads to an increased ability to pack local variable loads and stores, since the corresponding instructions are already available in the IRF and the IMM can supply necessary parameters. Additionally, the IMM allows us exploit our transformations for increasing the redundancy of semantically equivalent operations.

Positional register specifiers do not have a profound impact on average code size, but they still provide a measurable benefit. Individual benchmarks such as *blowfish* benefit by up to a 2% additional reduction. Other benchmarks like *adpcm* experience up to a 1% code size increase by using positional register specifiers. These slight increases are due to the greedy heuristics we apply in selecting instructions to reside in the IRF.

4.2. Energy and Execution Time Analysis

A modified version of the *sim-panalyzer* simulator is used to gather the energy consumption data for the benchmarks [23]. *Sim-panalyzer* calculates approximations of area size and number of gates for each component and ties

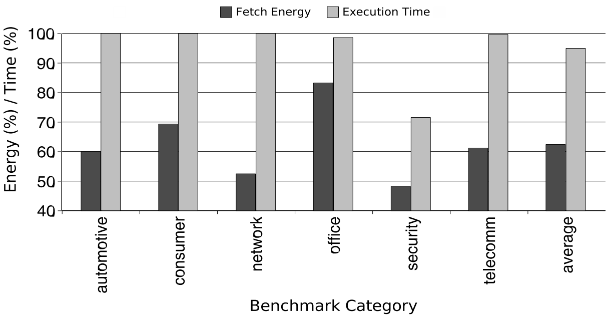


Figure 14. Reducing Fetch Energy/Exec. Time

these numbers to the cycle-accurate simulation provided via SimpleScalar. We modified the original Alpha port of *sim-panalyzer* to match the MIPS. The resulting energy estimates are not exact, but are accurate enough to support our claims.

The energy savings come from two sources. First, applications complete in fewer cycles due to the increased fetch rate that the IRF provides. Second, there are fewer accesses to the IC and memory as approximately 55% of instructions are fetched from the IRF. An access to the IC has a two orders of magnitude higher energy cost than an access to a register file. A memory access is another two orders of magnitude more costly than an IC hit.

Figure 14 shows the energy consumed during instruction fetch compared to the unpacked versions of the programs. On average, we obtain a 37% reduction in energy consumed by I-Fetch. Much of this savings comes from a reduction in number of IC references, with additional savings from a reduction in IC misses for some of the network and automotive applications and a reduction in execution time for the security applications. For these applications, I-Fetch energy comprises 30% to 45% of the total power requirements.

In most benchmarks, the improvements in execution time are minimal, executing in 98% to 100% of the original number of cycles. In no case did we incur a performance penalty. Average execution time savings was 5.04%, due primarily to three of the security benchmarks: *blowfish*, *pgp* and *rijndael* executed in 59%, 61% and 67%, respectively, of their original cycles. After instruction packing, the working set of these applications fits better into cache, and they have far fewer IC misses. The performance gains, though small for many of the benchmarks, also allow the programs to finish in fewer cycles, and this contributes to the overall energy reduction. Although the collected measurements are for fetch energy, the fetch power consumption results are similar for 5 of the 6 categories, since the execution times are approximately the same. Due to the improved execution time of many security benchmarks, their fetch power savings would be proportionally lower than their overall fetch energy savings.

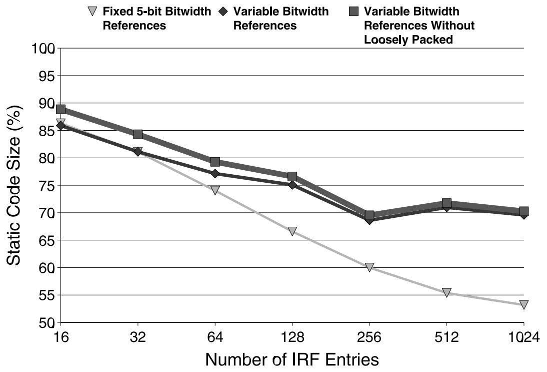


Figure 15. IRF Static Code Size Sensitivity

4.3. Varying IRF Size

To measure the scalability of the IRF scheme, we look at various sizes for the actual IRF. Of course, increased sizes for the IRF means that the available packing patterns could also vary. For example, with 16 IRF entries, we could potentially create a *tight6* pack, since it would only require 4-bit reference fields. With larger IRF entries like 128, we could only access the first 32 entries in a loosely packed instruction, and we could only support up to the *tight3* pack, since we require 7-bit IRF references. The lack of these larger packed formats could potentially decrease code density, even though a wider variety of instructions are available via the IRF. Additionally, power consumption and context-switching overheads could increase with a larger IRF.

Figure 15 shows the results of varying IRF size. Each scheme uses 32 entries in the IMM (except 16-entry IRF which uses 16), as other sizes do not provide much improvement. The fixed 5-bit bitwidth scheme models an ideal case where each IRF size can support all original packing formats arbitrarily. This better-than-ideal case assumes that we have some mechanism for allowing all IRF references to be encoded with only 5 bits for each slot. The variable bitwidth scheme is the actual case, where each size is modeled appropriately considering bitfield requirements. The final scheme is the variable bitwidth model with no loosely packed instructions. This case presents a view of the benefit obtained via loosely packing instructions. Considering that loosely packing instructions requires more ISA modifications than just adding opcodes for tightly packed instructions, the benefit may not outweigh the cost for some existing ISAs. However when designing a new ISA, it is important to consider the effects of adding loosely packed instruction formats as a flexible mechanism for improving code density.

Each time the size of the IRF is doubled, the compression improves by approximately 5%. Compression gets worse when moving to the 512 entry IRF, due to the elimination of the *tight3* packed instruction format that can no longer be referenced with the appropriate number of bits in a 32-bit MISA instruction. When moving from 32 to 64 and 64 to 128 entries, the effect is reduced, but we observe a similar trend since we lose some IRF reference fields.However these cases still retain other dense tightly packed instruction formats, and thus can provide improved compression ratios. Similar to other code compression techniques, there are diminishing returns when increasing IRF size.

5. Crosscutting Issues

Using an IRF and IMM that is unique to each process means that more state must be preserved at context switches. A conventional context switch simply saves state to memory at the point the switch occurs, and restores this state when the process resumes. One approach to reducing this overhead is to save a pointer to a routine associated with each process, where the routine will reload the IRF when a process resumes. Thus, the IRF would not need to be saved, only restored. However, positional register specifiers would need to be saved and restored on a context switch.

A related issue is how to address exceptions that occur during execution of a packed instruction when structural or data hazards prevent all of the instructions from being simultaneously issued. One or more RISA instructions referenced by the packed instruction have already completed at the point the exception occurs. One solution is to store in the context state how many RISA instructions within the packed instruction have completed, since these instructions are executed in order. A bitmask of retired instructions would allow for a precise restart after handling the exception in an out-of-order machine. The addition of a bitmask would not add much overhead to the system.

6. Related Work

Table 4 shows a comparison between prior research and the IRFs proposed in this paper. Entries are compared based on existing published research data regarding code size, power consumption and execution speed, as well as estimated hardware complexity, which includes hardware cost and design modifications. We also note cases where certain data was unavailable, and thus these values represent our best predictions based on the available information.

One of the early approaches to reduce code size and the cost of fetching instructions was microcode [25]. Each CISC or macro instruction fetched from memory caused a sequence of microinstructions to be fetched and executed, which provided a faster access time than main memory. Our proposed approach differs from microcode in several ways, including that specific instructions within the IRF can be individually referenced and that the instructions in the IRF can be changed for each executable.

Table 4. Comparison of Existing Techniques

Technique	Ref	Code Size Reduction	Power Savings	Speed	Hardware Complexity	Comments
Proc. Abs.	[10, 8, 5]	+	–	–	0	Additional function call and return penalties
L0	[13]	0	++	– –	0	Large execution time penalty for power savings
Echo	[16]	++	–	+/–	1	Better IC behavior but increase in exec insts
ZOLB/Loop Cache	[9, 11]	0	+	+	1	Improves speed slightly
IRF		++	++	+	**1**	Similar complexity as Register File + ZOLB
Codewords	[17]	++	–	?/– –	2	Decompression slows execution
Arm/Thumb	[22]	++	–	– –	2	Dual-width ISA trades speed for code size
Arm/Thumb/AX	[15, 21]	++	?/–	– –	2	AX reduces execution time penalty of Thumb mode
Heads and Tails	[19]	++	?/–	?/–	2	Better IC behavior but branch penalties
DISE	[6]	++	?/+	+	3	Complex decode can affect clock and pipeline depth
Mini-graphs	[4]	++	?/–	+	3	No power analysis, but additional ALUs

Legend: + means that improvement is $<$ 10%. ++ means that improvement is $\geq$ 10%. 0 means that there is very little to no effect. ? means that results are speculative since they are not presented or explained in detail. – means that penalty is $<$ 10%. – – means that penalty is $\geq$ 10%. Hardware complexity is scaled from 0 (no changes) to 3 (complete redesign).

Due to the increasing pervasive use of embedded systems, there has been a significant amount of recent work on compressing code. Some early work on code compression used a compiler optimization called procedural abstraction to reduce code size [10, 8]. Procedural abstraction can be viewed as the opposite of function inlining. Common code sequences are abstracted into routines, and the original sites of each sequence are converted into calls. Subsequent work involved register renaming to abstract more code segments [5]. The main disadvantage of procedural abstraction is that the program typically becomes slower due to the overhead of executing call and return instructions for each abstracted code segment.

Many code compression approaches use special hardware support to assist in the compression. One simple hardware extension is the echo instruction, which is in essence a lightweight procedure call [16]. The echo instruction indicates where the abstracted instructions can be found and the number of instructions to be executed. Unlike using a conventional procedure call to abstract a sequence of instructions, a return instruction is not needed at the end of the abstracted sequence. Another advantage of this approach is that abstracted sequences of instructions can overlap. However, there is still overhead due to transfers of control to the common code sequences, and reduced spatial locality for machines with a memory hierarchy.

Other approaches use a hardware dictionary, where duplicate code sequences are stored in a special control store in the processor, and codewords are associated with each of these sequences. One approach uses variable length codewords that align to a 4-bit boundary [17]. This approach has the disadvantage of complicating the I-Fetch and transfer of control logic, since instructions now vary in size. Compression can be further enhanced by considering opcodes and operands separately [1]. Yet another approach supports a parameterized hardware dictionary, where small differences can be abstracted by parameters that are encoded in the instruction containing the codeword representing the common sequence in the dictionary [6]. These approaches complicate the instruction fetch and decode, leading to potentially increased energy consumption and execution time. All of these approaches also required hundreds to thousands of instructions to be stored in the dictionary for compression to be as effective as our approach with only 32 instructions.

Another hardware code compression approach is to support dual instruction sets. Both the ARM/Thumb and MIPS16 architectures define 16-bit and 32-bit instruction sets, and the program can switch between the two using a special instruction [22, 14]. The 16-bit set has a reduced number of bits available to specify immediate values and registers. Using the 16-bit set typically saves space at the cost of additional instructions and increased execution times. Augmenting instructions (AX) extend the Thumb architecture, allowing the execution of two 16-bit instructions as a single 32-bit instruction [15]. This reduces some of the performance penalty in replacing 32-bit code with 16-bit code in a dual-width ISA. ARM's recently developed Thumb-2 incorporates some of the ideas behind AX to create a modified Thumb instruction set that supports both 16-bit and 32-bit instructions with increased expressiveness [21].

Heads and Tails advocates modifying the ISA for variable length instruction formats based on splitting the opcodes and operands of instructions [19]. Several instructions are bundled together with fixed length heads (describing opcodes and possibly a single register), and variable length tails which supply additional information (such as registers and immediate values). This scheme requires a significant modification of the standard MIPS ISA, yielding branch penalties that may increase power consumption and execution time, despite improvements to cache locality.

These compression approaches vary in effectiveness. Reported code size reductions on a variety of benchmark suites indicate that procedural abstraction achieves 5%-7%, echo instructions achieve 16%, hardware dictionaries achieve 33%-52%, and dual instruction sets achieve about 31%.

Heads and Tails reduces code size by up to 24.5%. However, all of these code compression approaches decrease code size at the expense of increasing execution time and/or require significant additional storage for the compressed sequences. Many of these approaches are complementary to instruction packing and could potentially be used in conjunction with an IRF to achieve even greater compression.

Zero overhead loop buffers (ZOLBs) are becoming fairly common in low-power embedded processors [9]. A ZOLB is essentially a compiler managed IC, where an innermost loop is explicitly loaded and executed. The advantages of using a ZOLB include elimination of loop overhead (increment, compare, and branch instructions) and reduced energy consumption due to instructions being fetched from the ZOLB instead of ROM. These loops are typically restricted in that: only a limited number of instructions fit in the buffer; there can be no transfers of control besides the loop branch; and the number of iterations the loop executes must be known. Thus, many parts of an application cannot execute from the ZOLB. L0 caches are another enhancement designed to improve power usage at the cost of execution time [13]. These caches are typically smaller than an L1 cache and direct mapped, yielding a low hit ratio, but a reduced energy-delay product. L0 caches can reduce power by 58% at the cost of a 21% performance penalty. Low-power embedded systems can also provide loop caches, similar to a ZOLB, that can be dynamically loaded if small-offset backward branches are detected [11].

Accessing registers positionally is similar to data movement in transport-triggered architectures (TTAs) [7]. These architectures provide a mechanism for explicit data forwarding, where explicit operations are specified to transport data between function units. TTAs are designed to reduce the number of ports required for the register file, while the IRF uses positional registers to allow more instructions to be packed. Dataflow mini-graphs also employ positional register specifiers in the construction of aggregate instructions [4]. Compiler analysis maps common operation sequences as complex new instructions via handles. This is similar to dictionary techniques for compression, however the sequences can be fetched and executed more efficiently by a superscalar architecture. Code size reductions are approximately 13% with a 512×3 entry mini-graph table, while execution time decreases by 7%. However, power consumption should increase due to additional ALUs as well as the mini-graph table structures.

7. Future Work

There are many alternatives to explore to increase the effectiveness of the IRF. These alternatives include changing the architecture, adding additional compiler optimizations, or both. We can dynamically load instructions into the IRF at various points in the execution to more effectively pack instructions in applications with widely varying phases. Similar to the SPARC register windows, we can explore the effects of greater IRF sizes using an approach where an IRF window pointer can be switched on function calls and returns [24]. Storing opcodes and operands separately has improved standard code compression, so it is likely that this approach would increase IRF packing flexibility [1]. Compiler optimizations can be developed to schedule instructions within a basic block to improve packing. We could determine a DAG of dependences between instructions in a basic block and use more aggressive packing algorithms to exploit alternative orderings. The IRF can also facilitate the design of new dual ISAs, where MISA instructions reduce code size and RISA instructions improve execution time.

8. Conclusions

We have explored using instruction registers to achieve significant code reduction without large dictionaries or complex instruction decoders. We find that a 32 entry IRF provides an average 19% reduction in code size for the embedded applications studied, and much greater compression for some of the larger applications. This surpasses the results of many earlier compression studies while avoiding much of the hardware complexity in the implementation and all of the performance loss found in the other schemes. Furthermore, code compression available via IRF encoding is orthogonal to some earlier compression techniques.

We expanded our study to include the impact of IRFs on processor power and execution time, and found additional benefits. A 32 entry IRF reduces energy consumption of I-Fetch by an average of 37%, which translates to an overall processor energy savings of 15% — and as much as a 45% for blowfish, which ran much faster due to a dramatic reduction in IC misses. Energy is saved because over 50% of all instructions were fetched from a very small, low-power register file instead of a larger IC. The utilization of the IC is also improved, lowering miss rate. We expect IRFs to be performance neutral, but consistently find a small savings (and in some applications, a large savings) due to a increased IC locality.

It is rare that a combination hardware/compiler optimization yields improvements in all three performance metrics. We believe we can further improve code compression and power savings through additional refinement of the algorithms and improvements in MISA and RISA formats. It may be possible to target performance directly in aggressive out-of-order, multiple issue pipelines by increasing the rate of I-Fetch after a branch misprediction recovery.

9. Acknowledgments

We are indebted to Krste Asanović and Sally McKee for many valuable discussions that improved the quality of this paper. 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] G. Araujo, P. Centoducatte, and M. Cortes. Code compression based on operand factorization. In *Proceedings of the 31st Annual Symposium on Microarchitecture*, pages 194–201, December 1998.

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

[3] 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.

[4] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 18–29. IEEE Computer Society, 2004.

[5] 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.

[6] 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.

[7] H. Corporaal. *Transport Triggered Architectures: Design and Evaluation*. PhD thesis, Delft University of Technology, September 1995.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[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] A. Krishnaswamy and R. Gupta. Enhancing the performance of 16-bit code using augmenting instructions. In *Proceedings of the 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, pages 254–264. ACM Press, 2003.

[16] 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.

[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] H. Pan and K. Asanović. Heads and Tails: A variable-length instruction format supporting parallel fetch and decode. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 168–175. ACM Press, 2001.

[20] D. Patterson and J. Hennessy. *Computer Organization and Design, The Hardware/Software Interface*. Morgan Kaufmann Inc., 1998.

[21] R. Phelan. Improving ARM code density and performance. Technical report, ARM Limited, 2003.

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

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

[24] D. Weaver and T. Germond. *The SPARC Architecture Manual*. SPARC International, Inc., 1994.

[25] M. Wilkes and J. Stringer. Microprogramming and the design of the control circuits in an electronic digital computer. In *Proceedings of the Cambridge Philosophical Society*, volume 49, pages 230–238, April 1953.