Adapting Compilation Techniques to Enhance the Packing of Instructions into Registers

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

ABSTRACT

The architectural design of embedded systems is becoming increasingly idiosyncratic to meet varying constraints regarding energy consumption, code size, and execution time. Traditional compiler optimizations are often tuned for improving general architectural constraints, yet these heuristics may not be as beneficial to less conventional designs. Instruction packing is a recently developed compiler/architectural approach for reducing energy consumption, code size, and execution time by placing the frequently occurring instructions into an Instruction Register File (IRF). Multiple IRF instructions are made accessible via special packed instruction formats. This paper presents the design and analysis of a compilation framework and its associated optimizations for improving the efficiency of instruction packing. We show that several new heuristics can be developed for IRF promotion, instruction selection, register re-assignment and instruction scheduling, leading to significant reductions in energy consumption, code size, and/or execution time when compared to results using a standard optimizing compiler targeting the IRF.

Categories and Subject Descriptors

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

General Terms

Performance, Power, Experimentation

Keywords

Instruction Register File, Instruction Packing, Compiler Optimizations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'06 October 23–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-543-6/06/0010 ...$5.00.

1. INTRODUCTION

Modern processor designs often trade off regularity and orthogonality in the instruction set or microarchitecture to better meet design constraints. This requires compiler writers to rethink well understood optimization techniques in order to avoid performance bottlenecks caused by an idiosyncrasy of the processor. Unconventional designs are perhaps most prevalent in new embedded architectures, since they face the most stringent design requirements for power, code size, and in some cases execution performance. These architectures can also more easily exploit new instruction set encoding techniques to meet these goals. This requires the compiler writer to not only target a new set of instructions, but often a very different method of encoding instructions that may not be suited to the standard techniques used in code optimization.

One promising embedded architectural feature recently proposed is the Instruction Register File (IRF), which places the most common instructions in a small 32-entry register file [10, 12, 11]. A new instruction is added to the ISA that references up to five IRF entries in a single 32-bit instruction. Use of the IRF results in reduced energy consumption, since packed instructions can fetch their component instructions from the lower-power IRF instead of the instruction cache. Packing also results in decreased code size, since multiple instructions from the base ISA can now be represented in the same space as a single instruction. This feature also reduces the memory footprint of the application, leading to slight improvements in execution efficiency.

The compiler determines which instructions are promoted to the IRF. Prior research used dynamic profile data to determine the most frequently accessed instructions and made minor changes to the instruction selection heuristics in the compiler [10, 12]. This approach enabled the IRF to significantly improve code size as well as energy consumption, but since the instruction selection, register allocation, and code scheduling transformations used in the compiler were not tuned to the requirements of an IRF architecture, there was still room for improvement. In fact, effective use of the IRF is more dependent on well-tuned compiler optimizations than a more conventional architecture. Also, the IRF approach provides some unique opportunities for code optimization that are counter-intuitive to those familiar with more conventional processor designs.

This paper demonstrates how existing compiler optimizations can be modified to improve the efficiency of instruction packing with an IRF. Enhancing these optimizations results in reduced fetch energy consumption, decreased static code size, and slight improvements in execution efficiency. This paper makes the following contributions:

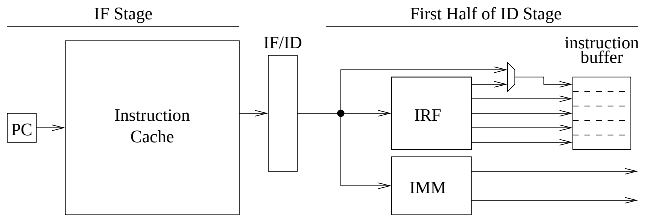


Figure 1: Decoding a Packed Instruction

- We provide the first detailed description and evaluation of a compiler framework for instruction packing.
- We propose and evaluate several enhancements for the promotion of instructions to the IRF. These enhancements include more accurately modeling the benefit of promoting an instruction, allowing additional I-type instructions to be promoted with different default immediate parameters, and integrating static and dynamic measures for selecting the instructions to promote. Mixed profiling allows a developer to fine-tune the characteristics of an application across several design constraints including static code size and overall processor energy consumption.
- We adapt existing transformations such as instruction selection, register re-assignment, and instruction scheduling to enhance the compiler's ability to pack instructions together. Our results show that these enhanced optimizations can significantly reduce both the static code size and energy consumption of an application, while providing a slight performance improvement.

2. INSTRUCTION PACKING

There are a multitude of techniques available for reducing energy consumption, decreasing code size, and improving execution time. However, these techniques often require tradeoffs between different design constraints that limit their effectiveness, since embedded devices often have to meet very rigid guidelines. Compression techniques [7, 6, 4, 15, 18] can reduce code size and small, specialized instruction caches [13, 8] can reduce energy consumption, but each can increase execution time. Similarly, increases in clock frequency can improve execution time, but can negatively impact the overall processor energy consumption. Instruction packing is a combination architectural/compiler technique for targeting all of these constraints simultaneously. Furthermore, instruction packing can be used in a complementary fashion with other compression and energy saving techniques, such as L0 (filter) caches [13] and loop caches [14].

The motivation for instruction packing is to keep frequently accessed instructions in registers, just as frequently used data values are kept in registers by the compiler through register allocation. Similar to the data register file, effective use of an IRF can have a dramatic impact on energy consumption, code size and execution efficiency. 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 (ID) stage. It is also possible to place the IRF at the end of instruction fetch (IF) or store partially decoded instructions in the IRF should the decode stage be on the critical path of the processor implementation.

6 bits	5 bits	5 bits	5 bits	5 bits	1	5 bits
opcode	inst1	inst2	inst3	inst4 / param	s	inst5 / param

Figure 2: Tightly Packed Format

Figure 2 shows the special MISA instruction format used to reference multiple instructions in the IRF. These instructions are called *tightly packed* since multiple RISA instructions are accessible via 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. The last two fields that could reference RISA instructions can alternately be used to reference immediate values from the IMM or a destination register number to replace the default immediate or destination register number of the RISA instruction, respectively. The number of parameterized values used and which RISA instructions will use them is indicated through the use of four opcodes and the 1-bit S field. Prior work with the IRF has used a profiling pass to determine the 31 most frequently referenced dynamic instructions to be 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.

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

Figure 3 shows the differences between the traditional MIPS instruction formats and the loosely packed MISA extension. With R-type instructions, the *shamt* field can be used for a RISA reference and shift amounts can now be specified in *rs*. Immediate values in I-type instructions are reduced from 16 bits to 11 bits to make room for a RISA reference. The *lui* (load upper immediate) instruction is the only I-type that is adjusted differently, in that it now uses only a single register reference and the remaining 21 bits of the instruction for the upper immediate portion. This is necessary since we still want a simple method for creating 32 bit constants using the *lui* with 21 bits for an immediate and another I-type instruction containing an 11 bit immediate value.

In order to more effectively pack instructions for applications with diverse function and phase behavior, the IRF was extended to support 4 hardware windows [12], much in the same way that the SPARC data register file is organized [22]. Using windows is preferable to just increasing the size of the IRF, since the windows do not require any changes to the tightly packed instruction format. This means that instead of using only 32 instruction registers, there are a total of 128 available physical instruction registers. Only 32 of these registers are accessible at any single point in time, however, so the remaining 96 registers can be kept in a low-power mode in which they retain their values, but cannot be accessed. On a function call and/or return, the target address uses 2 bits (shown

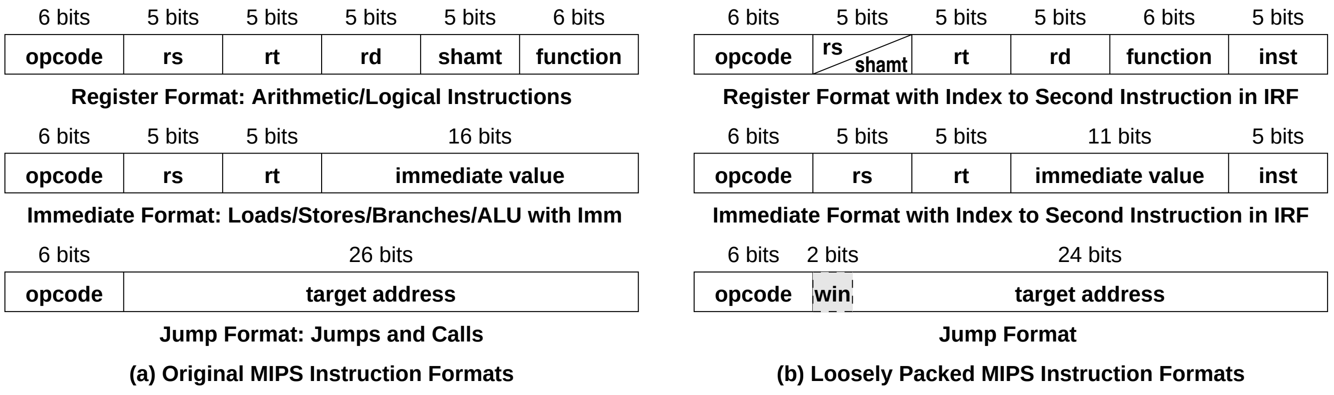


Figure 3: MIPS Instruction Format Modifications

as win) to distinguish which instruction window we are accessing. All function addresses are updated at link-time according to which window of the IRF they will access. The IMM for each window is the same, since previous results have shown that 32 immediate values are sufficient for parameterizing most instructions that will exist in an IRF. Using two bits to specify the instruction window in a function address limits the effective address space available for an application, but we believe that over 16 million instruction words is large enough for any reasonable embedded application.

Instruction packing, however, is not without its limitations. First of all, there is only so much redundancy available in the instructions of an application. Similar to existing code compression enhancements [5], support has been added in the form of parameterization to more effectively capture instruction redundancy. Second, the IRF can only hold a limited number of instructions, since IRF specifiers have to be encoded within the base ISA, and it is preferable to keep the overall complexity low for energy efficiency. IRF windows provided a convenient solution that minimized ISA changes, while also providing a simple mechanism for scaling IRF utilization.

These solutions have been primarily architectural in nature, and would be more difficult to address from the compiler perspective. Yet there still exist several limitations that can potentially be addressed by compiler optimizations. Existing instruction packing algorithms have focused primarily on the dynamic behavior of an application in an effort to minimize fetch energy [12, 11]. The static composition of an application should also be used when promoting instructions to the IRF, as code size can have a significant impact on the memory architecture of an embedded system. For an instruction to be promoted to the IRF, the opcode and all operands must match exactly. Parameterization provides a partial solution for capturing additional instructions, but the compiler can intelligently seek out even more cases where register operands can be modified to allow an instruction to be packed. Additionally, the order of instructions in a basic block can artificially limit the packing density, since packable instructions work best when they are adjacent. Finally, there are several artificial limitations on forming packs of instructions. For example, in previous implementations, any packed branch instruction had to occupy the final slot of the packed instruction, as packs were not allowed to span basic block boundaries. Each of these limitations can be effectively addressed with detailed compiler analyses and transformations.

3. IMPROVING THE PROMOTION OF INSTRUCTIONS TO THE IRF

Instruction promotion is the process of selecting which instructions should reside in each IRF window, as well as which immediate values should reside in the IMM. This process of promotion has been performed offline by supplying static or dynamic profile data to *irfprof*, an IRF selection and layout tool [12]. Functions are partitioned and placed into statically allocated IRF windows by irfprof according to a greedy algorithm that has been previously explored. This algorithm operates by estimating the potential cost/benefit of packing the instructions of a function into each particular IRF window, and then greedily selecting the most beneficial function to assign to a window until each function has been allocated.

In the original irfprof, IRF-resident instructions were evaluated as having a cost of 1, while non-resident instructions had a cost of 100. These costs were chosen based on the relative energy benefit of placing an instruction into the IRF versus keeping it in the instruction cache. However, not all instructions will obtain equal benefits from being promoted to the IRF. Parameterizable I-type instructions were originally coalesced together with the most frequent immediate value becoming the default immediate, while each individual parameterizable form of the instruction contributed to the overall benefit for promoting this instruction. However, the benefit of immediate instructions that require a parameter should be lower, since they will occupy two slots of a MISA instruction, thus making them impossible to loosely pack. They also require two register reads, since both the IRF and the IMM will be active when fetching this particular instruction.

The benefit of promoting to the IRF can be modeled more accurately by quantifying the possible potential improvement (based on code size and fetch energy). For instance, a tightly packed instruction cannot achieve any further benefit, so its potential improvement is 0. A parameterized packable instruction (one which has to use the IMM) has a potential improvement of 1, since it could be promoted with its immediate value as the default. A loosely packable instruction has a potential improvement of 3, since it normally would occupy approximately 4 of the slots in a MISA instruction, with the remaining slot available for a single RISA reference. Finally, an instruction that is not loosely packable like *lui* has a potential improvement of 4, since packing it into a single RISA entry will free up 4 additional slots in the MISA instruction. By calculating the potential improvements in this manner, we provide a means for multiple I-type instructions that differ only in default immediate value to reside in the IRF simultaneously. This allows each entry to remain loosely packable, which can be beneficial if each operation occurs very frequently.

In all prior IRF work, instructions have been promoted to the IRF based purely on static or dynamic profile data. Although the IRF is designed to improve the overall efficiency of instruction fetch, this division may not produce an adequate balance between code size savings and energy reduction, particularly when dealing with the highly constrained embedded design space. Dynamic profiling exposes the kernel loops of the application, and correspondingly the most frequently executed instructions from these loops. The static profile will likewise reveal those instructions that comprise the greatest portion of the application's code. A unified approach encompassing static and dynamic measures may yield a majority of the benefits of each, resulting in a more suitable packing strategy for the embedded domain. The promotion algorithm can be modified to incorporate the scaling of both the static and dynamic profile data to provide such flexibility.

4. INSTRUCTION SELECTION

Instruction selection is the process by which a compiler chooses which instruction or instruction sequence to use for a particular semantic operation. The VPO compiler operates on register transfer lists (RTLs) that have a one-to-one correspondence with machine instructions. We can modify instruction selection to increase the amount of redundancy in the code without negatively impacting code size or performance. There are several methods for using instruction selection in this manner. First, we can choose equivalent parameterizable operations to replace simple operations, such as encoding move operations as additions with 0. Second, commutativity rules can be applied to make sure that all semantically equivalent instruction instances use the same order for operands. Third, we can apply parameterization to the destination registers of R-type instructions, which were previously unable to be parameterized.

Choosing equivalent parameterizable instructions over simple instructions is a technique that has previously been applied to instruction packing [10]. In this paper, we quantify the exact benefits of these transformations in increasing the instruction redundancy within an application. Most of these equivalence transformations occur for the *mov* and *li* pseudo-instructions. Register moves are normally performed using the *addu* instruction with the hard-wired register zero as the second source argument. Instruction selection instead generates this operation as an *addiu* instruction with zero as the immediate operand. Load immediate instructions with small constants can interchangeably be generated as *addiu* instructions or *ori* instructions that use register zero as their first source operand. To increase code redundancy, the profiling pass always converts these instructions to an *addiu* format. Each of these transformations increase the number of opportunities that parameterization will have for packing various sequences of instructions.

Simple transformations can also be used to increase redundancy by reducing or completely eliminating instruction diversity. The native MIPS ISA uses PC-relative addressing for branches and absolute addressing for jumps. Absolute addressing poses problems with instruction packing, since there can be quite a diverse set of jump target addresses. To increase the ability for frequent jumps to be placed in the IRF, short distance jumps (-16 to +15 instructions) are converted into branches that compare register zero to itself. These instructions can then be parameterized in the same manner as conditional branches. If short distance jumps occur frequently in the application, then only a single RISA entry is necessary to parameterize each of them.

The prior ISCA work also applied transformations to place the operands for commutative operations in the same order for each instruction. If the destination register is also a source register, then that register is placed first in the operand list. If all registers are different, then the operands are ordered from lowest to highest number. This transformation unifies equivalent commutative operations in an attempt to further increase the level of instruction redundancy.

Although parameterization of I-type RISA instructions has always been available to the IRF, in this paper we have extended simple parameterization to R-type destination registers. This works similarly to traditional IRF parameterization, consuming an additional RISA slot in the tightly packed instruction format to specify the replacement value (5 bits) for *rd*. It is important to note that the requirements for supporting this feature are minimal, as the existing parameterized instructions will not require any modifications. Only a small amount of additional hardware is necessary, primarily in the form of multiplexers going to the instruction decoder.

5. REGISTER RE-ASSIGNMENT

Compilers often attempt to minimize register usage in order to keep additional registers available for further optimizations. Since the VPO compiler applies optimization phases repeatedly, it also rigorously attempts to minimize the number of distinct registers used in each particular function. This strategy can clearly lead to different register usage patterns in the generated code for similar but slightly different functions due to the varying register pressure. A small difference in register numbering can eliminate the possibility of instruction packing for a sequence of instructions. Although the IRF supports a limited ability to parameterize registers, register re-assignment can be beneficial by replacing entire register live ranges. With re-assignment, these registers can be adjusted to match existing IRF instructions, leading to increased pack density.

Optimizing compilers have often employed register renaming to eliminate anti-dependences in generated code [17, 21]. Anti-dependences restrict the scheduling of instructions for an in-order pipeline, and can also negatively affect the issue of instructions in out-of-order pipelined architectures. It is for this reason that many modern out-of-order architectures employ additional hardware register renaming techniques to eliminate anti-dependences. Rather than renaming to avoid anti-dependences, we will re-assign registers to make instructions match existing IRF entries when possible.

Although compiler register renaming algorithms often operate within basic blocks to keep compile time fast, the IRF register re-assignment algorithm uses a register interference graph to calculate the entire inter-block live range span for each register. When constructing the register interference graph, registers that are used and set within a single RTL are split into two distinct live ranges. This splitting allows us to re-assign registers in a more fine-grained manner than the merging of these live ranges would have allowed. Shorter live ranges have reduced potential for conflicts, which can limit the effectiveness of such a transformation.

We use a greedy algorithm for selecting the candidates for register re-assignment. Basic blocks are ordered from most frequently executed to least frequently executed based on dynamic profiling data. With this information, each potential re-assignment is examined individually. Live ranges of registers that cannot be altered (e.g. calling conventions) are marked so they are not re-assigned in any manner. Since we are not going to perform multiple renames simultaneously, we must verify that the target register to which we are attempting to re-assign is not live at any adjacent node in the graph. Using the register interference graph, we can now perform the register substitution on the appropriate portion of each given RTL. Note that we cannot change all references, since we are splitting uses and sets within a single RTL into multiple live ranges of the same register number.

A) Instructions		**B) Packed Instructions**	**C) Re-assigned Instructions**		**D) Packed Instructions**
.L164:		.L164:	.L164:		.L164:
lw **$8**,0($14)	#	**lw $8,0($14)** {**2**}	lw **$2**,0($14)	# **IRF (1)**	**param4d**{**1,2,3,4,.L165**}
lw $3,0($12)	# IRF (2)		lw $3,0($12)	# IRF (2)	
slt $1,$3,**$8**	#	slt $1,$3,**$8**	slt $1,$3,**$2**	# **IRF (3)**	
beq $1,$0,.L165	# IRF (4) +	beq $1,$0,.L165	beq $1,$0,.L165	# IRF (4) +	
	# IMM (.L165)			# IMM (.L165)	
sw $3,0($14)	#	sw $3,0($14)	sw $3,0($14)	#	sw $3,0($14) {**7**}
sw **$8**,0($12)	#	sw **$8**,0($12)	sw **$2**,0($12)	# **IRF (7)**	
.L165:		.L165:	.L165:		.L165:
...		...	...		...
bne $1,$0,.L164	#	bne $1,$0,.L164	bne $1,$0,.L164	#	bne $1,$0,.L164

Figure 4: Register Re-assignment

Figure 4 shows an example of register re-assignment. The code is a single loop with an if statement guarding two store instructions. Column A shows the component instructions in the code sequence along with relevant data regarding the IRF entry numbers of the packable instructions. Note that the IRF contents are already determined at this point, and any unmarked instruction is not available via the IRF. The overall packing of the entire loop, assuming that no other transformations are applied, is shown in column B. If register re-assignment is performed on the code, then we obtain the code shown in column C. The last column (D) shows the re-assigned code after packing the instructions. The result is that the first two blocks of the original loop that required five MISA instructions can now be accomplished in two MISA instructions.

6. INSTRUCTION SCHEDULING

Instruction scheduling is another traditional compiler optimization that reorders the instructions in a basic block in an attempt to eliminate pipeline stalls due to long operation dependences. The actual scheduling often employs a directed acyclic graph (DAG) to maintain instruction dependence relationships. Once the DAG is constructed, instructions are issued based on priorities relating to future dependences. Instructions that have no incoming arrows in the DAG are considered to be in the *ready set*, as they have no dependences on which to wait.

Packing multiple RISA instructions into a single MISA instruction is somewhat similar to very-long instruction word (VLIW) scheduling. In addition to physical hardware constraints, the instructions in a VLIW word are executed simultaneously, so dependences have to be placed in separate VLIW words, leading to a great deal of fragmentation. Scheduling for IRF is similar to VLIW instruction scheduling, but the primary difference is that dependent instructions can be packed together in a single pack, since the individual RISA references will still be sequentially issued.

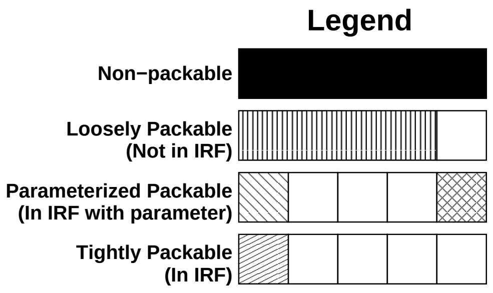
Figure 5 shows the algorithm for scheduling IRF instructions within a basic block. This greedy algorithm is based on several heuristics for producing dense sequences of packed instructions. It is invoked iteratively using the ready set until all instructions have been scheduled for the current block. It is important to note that the ready set from which selection occurs is sorted with respect to minimizing stalls due to instruction dependences. Thus, the dependence between instructions often acts as the tie-breaker for selecting which IRF or non-IRF instruction should be scheduled next. Priority is primarily given to loose packs between instructions that do not exist in the IRF and tightly packable RISA references. If three or more RISA reference slots (both IRF instructions and parameters) are available, then a tightly packed instruction will be started instead. When issuing into a started tightly

```
if blk→slots == 0 then
    if blk→prev_packable and an IRF insn is in ready set then
        schedule the IRF insn next
    else if 3 or more IRF slots are in ready set then
        schedule an IRF_param or IRF insn next
    else if IRF insn and packable insn are in ready set then
        schedule a loose pack with the packable insn and the
        IRF insn
if no instructions have been scheduled yet then
    if (blk→prev_packable or blk→slots == 4) and an IRF insn exists
    in ready set then
        schedule the IRF insn next
    else if 1 ≤ blk→slots ≤ 3 and any IRF insns in ready set then
        schedule an IRF_param or IRF insn next
    else if A non-IRF insn exists in ready set then
        schedule the non-IRF insn
    else
        schedule the first IRF_param or IRF insn next
Update blk→prev_packable and blk→slots based on the
scheduled instruction(s)
```

Figure 5: Intra-block Scheduling

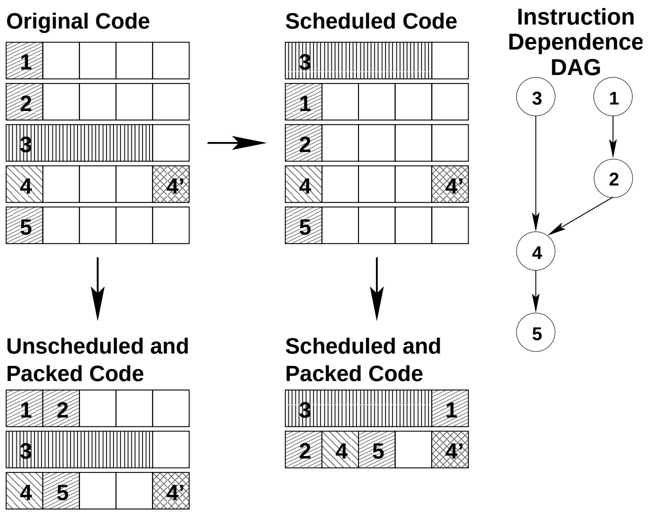
packed instruction, we always attempt to schedule the parameterized references first, since they require two slots and are unable to be loosely packed. If we cannot schedule into a loose pack or a tight pack, then we attempt to schedule non-IRF instructions next. This allows us to potentially free up dependent IRF instructions for packing on future iterations. Finally, we schedule IRF instructions if there are no ready non-IRF instructions. After choosing an instruction or instruction sequence for scheduling, the *prev_packable* and *slots* fields in the basic block structure must be updated appropriately.

Figure 6 shows the breakdown of instruction types used in the diagrams for the remainder of this section. Colored boxes refer to used portions of the instruction format. Empty boxes denote unused RISA slots. Non-packable refers to instructions that cannot support a loosely packed RISA reference and are not available via the IRF themselves (e.g. *jal*). A non-packable instruction occupies the space for all 5 RISA slots, and so there are none available for packing. Loosely packable refers to an instruction that is not available via the IRF, but has additional room for a RISA reference. These instructions occupy 4 of the 5 RISA slots, and so can accept a single non-parameterized IRF instruction. The parameterized tightly packable instruction is one that is available via a combination of the IRF and parameterization. The parameter can refer to an entry in the IMM table, a short branch/jump offset, or register parameterization. Due to referencing both the IRF entry



Legend
Non−packable
Loosely Packable (Not in IRF)
Parameterized Packable (In IRF with parameter)
Tightly Packable (In IRF)

Figure 6: Instruction Scheduling Legend



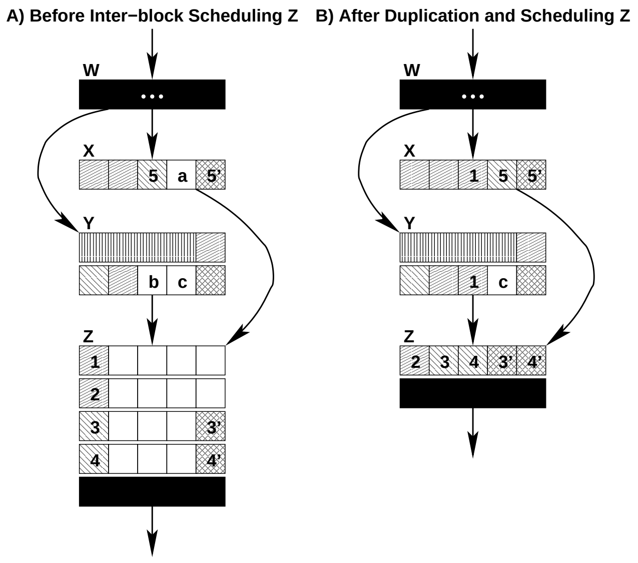
Original Code
1
2
3
4
4'
5
Scheduled Code
3
1
2
4
4'
5
Instruction Dependence DAG
3
1
2
4
5
Unscheduled and Packed Code
1 2
3
4 5 4'
Scheduled and Packed Code
3 1
2 4 5 4'

Figure 7: Intra-block Instruction Scheduling

and one IMM entry, two slots are occupied, and thus there is space for up to 3 additional RISA references. Tightly packable refers to an instruction that is available in the IRF, and does not require any parameterization. These instructions will occupy only a single slot, and thus have room for up to 4 more RISA references.

Figure 7 shows an example of intra-block instruction scheduling for improved packing efficiency. The original code consists of five instructions, of which three are in the IRF (1, 2, 5), one is in the IRF with a parameter (4), and one is loosely packable, but not available in the IRF (3). Based on the initial packing algorithm and no scheduling, we can only pack this sequence down to three total instructions, since instruction 3 cannot be combined effectively with any of its neighboring instructions. Since our algorithm favors loose instruction packs, instructions 1 and 3, which are both ready at the start of the block, can be combined into a single loosely packed MISA instruction. Instructions 2, 4, and 5 can then be combined into a *param3b* instruction. With the intra-block scheduling, we can shorten this sequence down to two total instructions, leaving only a single IRF slot empty.

Although conventional instruction scheduling may not include transformations that move instructions across basic blocks, IRF packing can benefit from inter-block scheduling. Instructions are packed using a forward sliding window and thus the final instructions in a block can be left with unused IRF slots. Although intra-block scheduling is an attempt to reclaim unused RISA reference slots, there are two cases where inter-block movement of instructions can lead to improved pack density. The first case is duplicat-



A) Before Inter−block Scheduling Z
B) After Duplication and Scheduling Z
W
...
X
5
a
5'
Y
b
c
Z
1
2
3
3'
4
4'
W
...
X
1
5
5'
Y
1
c
Z
2
3
4
3'
4'

Figure 8: Duplicating Code to Reduce Code Size

ing code for an unconditional successor block in each predecessor. Typically code duplication only serves to increase code size, but packed instructions that lead off a basic block can potentially be moved into unused slots in each predecessor. The second improvement is the addition of instructions after a packed branch, which will be described later. Each of these inter-block techniques attempts to more densely pack blocks that have already been scheduled. Although the code size may remain the same, by moving these operations earlier in the control flow graph (CFG), we are attempting to improve our ability to pack instructions in the current block. The proposed inter-block scheduling technique is quite similar to filling delay slots in a RISC architecture, particularly the annulled branch feature of the SPARC [22].

One interesting phenomenon with inter-block instruction packing is that duplication of code can lead to an overall code size reduction. Figure 8 shows an example of such a transformation on an if-then-else code segment. Basic blocks W, X, and Y have been scheduled, and block Z is about to be scheduled. Due to the number of tightly packable and parameterized packable instructions in Z, we know that the minimum code size (disregarding any dependencies) for this block must be three MISA instructions ($\lceil (4+2+5 \text{ slots})/5 \rceil$). We also notice that the two predecessors of Z (X and Y) have Z as their unconditional successor (fall-through or jump target). There are available RISA slots at the end of both basic blocks (slots a, b, c). Instruction 5, which occurs in block X is an example of a short jump instruction that has been converted to an unconditional branch with a parameter. Notice that for block X, the available slots are calculated without regard for the jump instruction, as the duplicated instruction will have to be placed before the jump in any case. Figure 8B shows instruction 1 after it has been duplicated in both predecessors of Z. It is able to be combined in two separate tight packs. Block X shows that the moved instruction is actually placed before the jump in order to maintain correctness. After performing intra-block scheduling on block Z, the parameterized instruction 4 is packed with instructions 2 and 3. This ultimately results in a net code size reduction of one instruction.

The baseline MIPS ISA that underlies our IRF architecture does not have support for predicated execution of instructions. With

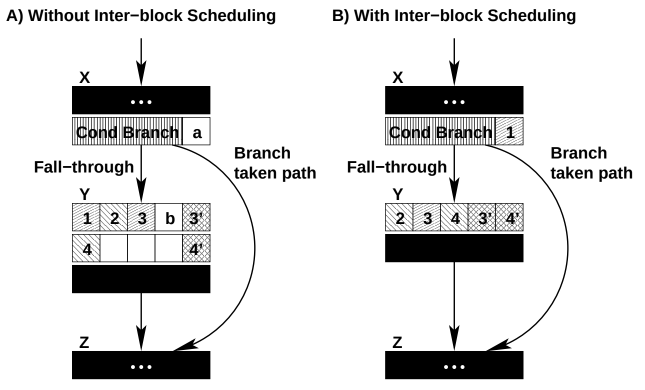


Figure 9: Predication with Packed Branches

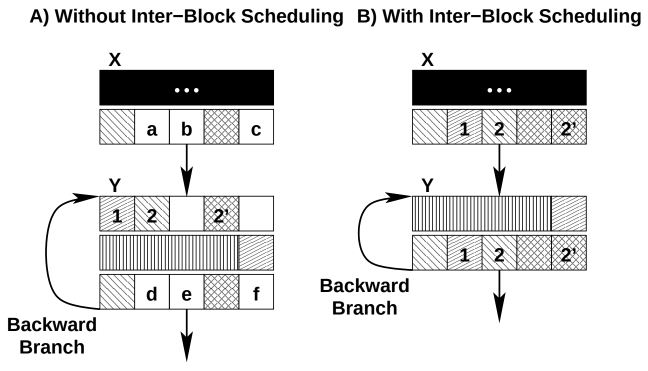

A) Without Inter−Block Scheduling
B) With Inter−Block Scheduling
X
a
b
c
Y
1
2
2'
d
e
f
Backward Branch
X
1
2
2'
Y
1
2
2'
Backward Branch

Figure 10: Backward Branch Scheduling

compiler transformations, however, we can mimic predication by packing instructions after conditional branches. If a forward conditional branch is taken, then the following instructions within the pack will be skipped. If it is not taken, then they will be executed normally, just as the fall-through block normally is. Backward branches are assumed to execute the additional RISA slots only when they are taken. The baseline IRF implementation reserves 5 bits for loosely packing each I-type instruction (except *lui*), and the original compiler did not support cross-block packing. Thus, branches could never loosely pack an additional instruction, and branches within tight packs always forced termination of the pack execution. This only serves to decrease the overall packing density. Note that we will not pack multiple branches or jumps together, since we still want the branch predictor and branch target buffer to be associated with the overall MISA instruction address. One benefit of this style of predicated execution, is that we do not require any additional bits in the traditional instruction formats for predicates. Furthermore, these predicated instructions need not be fetched, decoded or even executed if the predicate is false, which is not the case for other predicated ISAs like the ARM [20].

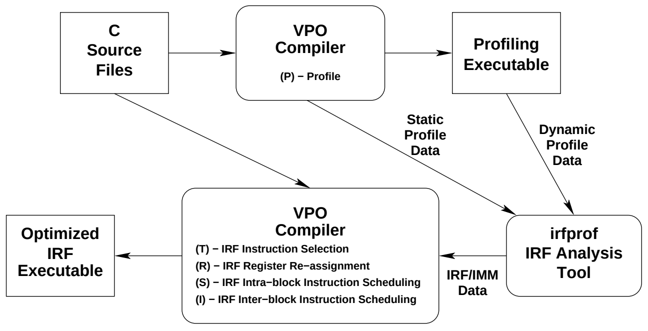
Figure 9 shows the potential benefits of predication using a simple if-then control flow built out of packed instructions. In Figure 9A, which does not have inter-block instruction scheduling, block Y consists of three MISA instructions, two of which are packed instructions, while its only predecessor (block X) contains a conditional branch with a target of block Z. The conditional branch in block X has one available RISA slot (a) for packing. Note that the RISA slot b is unusable since the parameterized instruction 4 requires two slots. In Figure 9B, which does perform inter-block instruction scheduling, instruction 1 is moved from block Y into the empty slot (a) of the conditional branch. This results in the ability for instructions 2, 3 and 4 in block Y to be packed efficiently into a single tightly packed instruction. This results in a net code size savings of one instruction.

Figure 10 shows an example of how instruction scheduling is used to improve pack density in the case of a backward branch. In Figure 10A, block Y consists of 3 MISA instructions including a backward branch back to the top of the block, while the preceding block X has a parameterized packable final instruction. The pack containing the backward branch in block Y has 3 available slots (d, e, f), and block X has 3 extra slots as well (a, b, c). Since the branch in Y is backwards, any following RISA entries will be executed only when the branch is taken. Thus, we can move instructions 1 and 2 (along with its parameter 2') into both the loop preheader (a, b, c) and the tail of the loop (d, e, f), as shown in Figure 10B. This movement of instructions is reminiscent of software pipelining, although additional registers are unnecessary for carrying the loop dependencies. After performing this optimization, we can see that the code size has been reduced by one MISA instruction. This transformation would be performed even if slots were unavailable in the preheader. The total code size would be the same in this instance, but the number of dynamic MISA instructions fetched would be reduced since the number of MISA instructions in the loop has been decreased.

The complete instruction scheduling algorithm for improving pack density is shown in Figure 11. It starts by performing intra-block scheduling on the function entry block and all loop headers. We then proceed by choosing the next block that has each of its predecessors already scheduled. If such a block is not found, then we select the next un-scheduled block and perform just the intra-block scheduling pass. If all predecessors of a block have been scheduled, however, then we have an opportunity to perform inter-block instruction scheduling to move instructions from the current block up into each predecessor. We first check if this block has a single predecessor that ends with a conditional branch. If the last MISA instruction in the predecessor has available RISA slots, then we attempt to choose IRF instructions for movement into the available slots. If the block has multiple predecessors, we can attempt to do duplication. Each predecessor block needs to have already been scheduled, have additional slots, and have the current block as their unconditional successor or branch fall-through. At this point, IRF instructions can be moved from the current block back into each individual predecessor block. Any predecessor that is terminated by a jump will have the moved IRF instruction placed in front of the jump, since jumps automatically terminate basic blocks and packs. Each predecessor that has instructions moved into it is then re-scheduled locally in order to see if a better packing solution exists and more slots can be freed. After all inter-block scheduling has been done, the current block is locally scheduled. By performing the inter-block scheduling early, we are filling up slots in blocks that have already been scheduled. This has two benefits: reducing the number of instructions to schedule in the current block, and moving deeper, dependent instructions closer to being ready in the current block. These benefits will then allow the intra-block scheduler to do a better job of forming dense instruction packs. If this block contains a backward branch for a loop, then we attempt to move instructions into any additional slots after the backward branch. To do this, we have to examine all predecessors of the loop header to calculate the minimum number of available slots. At this point, we can move instructions from the loop into each predecessor block and reschedule.

```
irf_intra_sched(entry_blk)
foreach blk that is a loop header do
    irf_intra_sched(blk)
while all blocks have not been scheduled do
    blk = next block with all preds scheduled
    // Predication
    if blk is fall through from branch and has no other preds
    then
        if predecessor has empty slots after branch then
            attempt to move IRF insns from blk into the
            slots
    // Duplication
    ok = TRUE
    foreach pblk $\in$ blk$\rightarrow$preds do
        if pblk is unscheduled or pblk$\rightarrow$left $\neq$ blk
        or pblk has no slots then
            ok = FALSE
    if ok then
        slots = minimum of available slots from all pre-
        decessors
        foreach pblk $\in$ blk$\rightarrow$preds do
            attempt to move IRF insns from blk into the
            slots
            irf_intra_sched(pblk)
    irf_intra_sched(blk)
    // Backward Branch Packing
    if blk branches back to loop header toblk then
        slots = minimum of slots from toblk preds in-
        cluding blk
        foreach pblk $\in$ toblk$\rightarrow$preds do
            attempt to move IRF insns from toblk into
            the slots
            irf_intra_sched(pblk)
    mark blk as scheduled
```

Figure 11: Inter-block Scheduling



C Source Files
VPO Compiler
(P) – Profile
Profiling Executable
Static Profile Data
Dynamic Profile Data
Optimized IRF Executable
VPO Compiler
(T) – IRF Instruction Selection
(R) – IRF Register Re–assignment
(S) – IRF Intra–block Instruction Scheduling
(I) – IRF Inter–block Instruction Scheduling
IRF/IMM Data
irfprof IRF Analysis Tool

Figure 12: Optimizing for Instruction Packing

Table 1: MiBench Benchmarks

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

7. EXPERIMENTAL EVALUATION

Our modeling environment is an extension of the SimpleScalar PISA target [1] that was previously used to study instruction packing [10, 12, 11]. Each simulator is instrumented to collect the relevant data involving instruction cache and IRF access during program execution. The baseline IRF configuration has four windows of 32 instruction register entries and supports parameterization via a single, 32-entry immediate table. The relative improvement due to compiler optimizations is similar for non-windowed IRF configurations. We model an out-of-order, single issue embedded machine with separate 8KB, 4-way, set-associative L1 instruction and data caches and a 128-entry bimodal branch predictor. Power estimation is performed using version 1.02 of the Wattch extensions [3] for SimpleScalar. Wattch models the power requirements of individual structures within the pipeline based on Cacti [23] estimations. The total energy estimates presented in this paper are based on Wattch's aggressive clock-gating model (*cc3*). Under this model, power consumption is scaled linearly for active units, while inactive portions of the pipeline dissipate only 10% of their maximum power.

Code is generated using a modified port of the VPO compiler for the MIPS [2]. This is the same compiler used in previous IRF studies. Figure 12 shows the general flow of operations for compiling code to support instruction packing with an IRF. Each benchmark is profiled statically and dynamically using SimpleScalar and then instructions are selected for packing using irfprof. The application is then recompiled and instructions are packed based on the IRF layout provided by irfprof. The optimizations that have been tested and applied are listed in the diagram. IRF register re-assignment occurs after IRF instruction selection. IRF instruction scheduling is performed after all other IRF optimizations. Note that instruction packing and the associated optimizations are performed only on the code generated for the actual source provided for each benchmark. Library code is left unmodified and is not evaluated in our results for static code size, however we do present total energy and execution time results based on the complete application behavior. If library code was subject to instruction packing, we would expect results to improve, since several benchmarks make extensive use of the standard C library functions.

In keeping with previous research on the IRF, we also selected the MiBench embedded benchmark suite for our experiments [9]. MiBench consists of six categories, each designed to exhibit application characteristics representative of a typical embedded workload in that particular domain. Figure 1 shows the benchmarks and associated categories evaluated in each of our experiments. Results are presented by category average or overall average in each of the following graphs in this section.

Instruction fetch has been shown to consume nearly one third of the total processor power of the StrongARM SA110 [16], so fetch energy efficiency can be extremely important for embedded systems. Figure 13 shows the results of applying these enhanced optimizations in terms of total processor energy. This is different from past work on the IRF which presents results for reducing only the fetch energy consumed by the processor. The baseline IRF architecture with no optimizations and dynamic profiling reduces total energy to 87.8% on average. Incorporating the enhanced mixed instruction promotion increases the total energy consumption to 87.99%, since we have traded some of our less important dynamic IRF entries for the ability to capture highly redundant static code. Instruction selection boosts the total energy efficiency, dropping the overall average to 84.35%. The re-assignment of registers increases the total energy to 84.71%, since it is focused primarily on improving static code size. Intra-block instruction scheduling is able to reduce the total energy to an average of 84.55%. Allowing for inter-block instruction scheduling further reduces the total en-

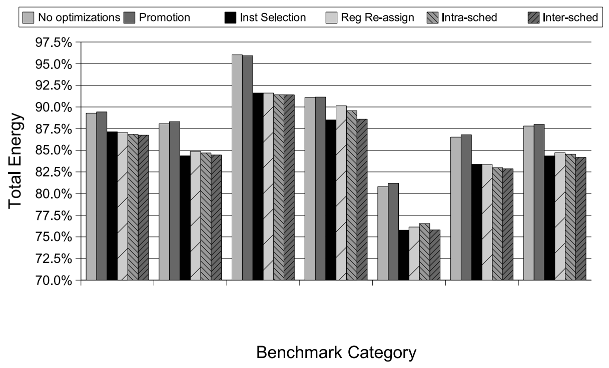


Figure 13: Total Processor Energy

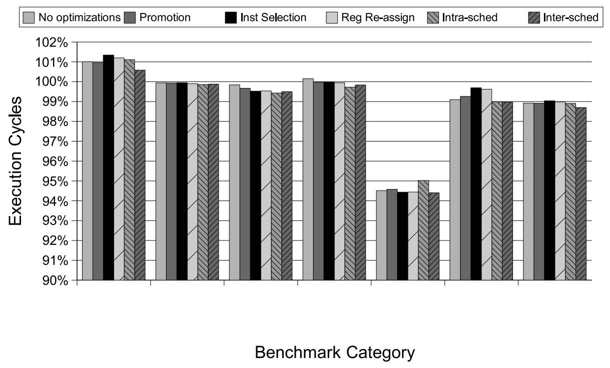


Figure 15: Execution Time

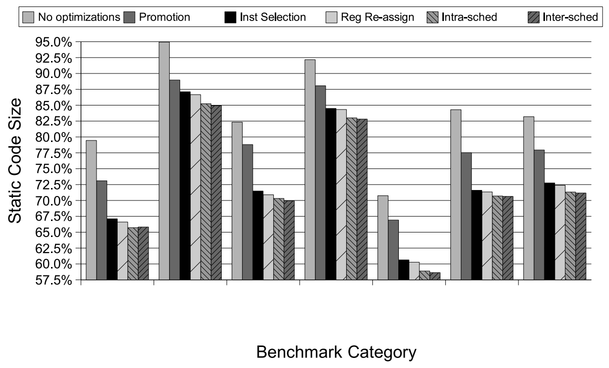


Figure 14: Static Code Size

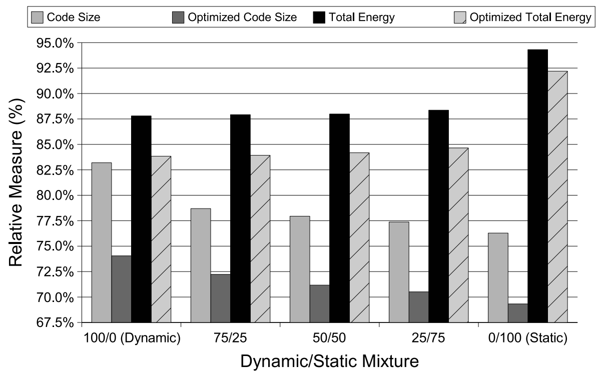


Figure 16: Evaluating Enhanced IRF Promotion

ergy consumption to 84.18%. Overall, instruction selection is the optimization that has the greatest impact on total energy reduction.

Figure 14 shows the resulting relative static code size of an application based on the application of these modified optimizations. The code generated for a non-IRF implementation of the ISA corresponds to 100%. The first bar shows the static code size for applying our IRF selection process to each application using only the 4-window IRF with immediate parameterization capabilities. The IRF with no optimizations is only able to reduce the code size to 83.20% on average, while supporting the enhanced mixed promotion drops the code size to 77.95%. After applying our instruction selection and register parameterization optimizations, the average code size is reduced to 72.76%. Applying register re-assignment reduces the code size to 72.39% of the original code size. Intra-block scheduling further reduces code size to 71.33%, while the addition of inter-block scheduling drops it to an average of 71.17%. These results are consistent with our intuition that the largest optimization benefits for code size would occur due to instruction selection and intra-block scheduling, however using a mixed dynamic and static profile also has a significant impact on code size.

The results regarding execution efficiency of the enhanced IRF optimizations are shown in Figure 15. The baseline IRF is able to reduce the execution time to 98.92% on average with dynamic profiling, and 98.91% for the enhanced mixed promotion. Adding instruction selection slightly increases the execution time to 99.04% Register re-assignment reduces the execution time to 98.98%, while intra-block instruction scheduling gets the overall execution time down to 98.90% on average. Inter-block instruction scheduling decreases the execution time to 98.7%. Fluctuations in execution time and/or IPC are primarily due to branch misprediction, which must now account for restarting in the middle of a packed instruction (e.g. predication). Improved profiling for commonly predicated instructions could help to make better decisions about which instructions should be moved across basic blocks. Additionally, instruction scheduling can sometimes move dependent instructions closer to their original dependence, leading to potential pipeline stalls. Several of the worse performing benchmarks are dominated by library calls. Packing these library functions specifically for these applications could lead to significant improvements in execution efficiency.

Each of the enhanced optimizations provided significant benefits for at least one benchmark. Enhanced promotion reduced the code size of Susan by an additional 16.30%. Instruction selection was responsible for an additional 10.73% reduction in code size for CRC32. Basicmath was reduced a further 1.67% in code size by register re-assignment. Intra-block scheduling can reduce the code size of Tiff by an additional 1.95%. Inter-block scheduling reduced the total energy consumption of Blowfish by another 2.78%. By providing each of these optimizations, we facilitate individual application improvements that are quite significant, particularly for the embedded domain.

Figure 16 represents a sensitivity study on the impact of combining static and dynamic profile information in various ways. Only code size and total energy are shown since the execution time benefits are similar to the previous experiments. Each combination of static and dynamic profile data is tested both with and without all of the previously described optimizations, although promotion enhancements are enabled for all experiments. As was expected, the static profile data yields the greatest code compression, while dynamic profile data yields the greatest total energy savings. It is interesting that almost all of the energy benefits (83.84% for optimized 100/0 and 84.18% for optimized 50/50) of promoting only the most frequent dynamic instructions can still be obtained while incorporating additional frequent static instructions in the IRF. In this case, the static code size can also be reduced from 74.05% for optimized 100/0 to 71.17% for optimized 50/50. This is reasonably close to the 69.33% relative code size that can be obtained from packing based on a purely static profile. The significance of the results of this experiment is that an embedded developer can adjust the balance of static and dynamic measures to meet varying design constraints regarding code size and energy consumption.

8. FUTURE WORK

The optimizations focused on in this paper happen relatively late in our optimizing compiler. It is possible that enhancing earlier stages of optimization can be even more beneficial for packing. Register assignment is the required phase in which physical register numbers are assigned to pseudo-register live ranges. This process happens fairly early in code optimization, as many other optimizations depend on knowing how many physical registers are still available (register allocation of variables, loop-invariant code motion). Our register re-assignment optimization is currently designed to remap register live ranges after the IRF contents have already been decided. We can also perform register re-assignment before profiling in an attempt to target specific registers for certain operations. By making certain registers more likely to be used for particular opcodes, the amount of redundancy in the code can be increased. Skewing the distribution of registers could lead to more saves and restores for callee-save registers, but the reductions in fetch energy consumption from increased packing density (particularly within tight loops) could outweigh the potential code growth.

Other compiler optimizations may enhance the opportunities for instruction packing. Applications with large basic block sizes favor instruction scheduling approaches, particularly if the basic blocks consist of mostly independent instructions. Loop unrolling transformations can often yield larger basic block sizes, by compacting multiple iterations of the same loop. The IRF can enhance loop unrolling by packing the duplicated loop body instructions, potentially using register parameterization. Profile guided code positioning algorithms cause blocks in the frequent path to have more fall-through transitions [19]. This can facilitate inter-block scheduling transformations that perform predication, since there will be many frequently executed fall-through transitions. Loop unrolling causes code size increases that may not be appropriate for embedded systems, however the IRF may be able to minimize the impact of the code duplication.

It may also be possible to improve the packing of instructions through the use of more flexible parameterization styles. Although the current implementation parameterizes the *rd* or immediate field of instructions, we can change the IRF instruction representation so that each entry specifies how a parameter is to be used. This would allow us to parameterize based on the *rs*, *rt* fields or a combination of fields (for accumulator style instructions). For example, we could implement a parameterizable increment instruction by specifying "*add $r3, $r3, 1*" as the IRF entry and *rs/rt* as the parameterizable portion.

9. CONCLUSIONS

The IRF architecture is capable of more efficient instruction encoding than conventional ISAs. In this paper, we evaluated a set of compilation techniques that have been adapted to exploit the IRF to further improve code density while minimizing instruction fetch energy requirements. We have shown that incorporating both static and dynamic measures for selecting instructions for IRF promotion can allow for the majority of the benefit that each could provide individually when targeting code size or energy reduction respectively. Instruction selection was modified to prioritize for the selection of the most common instruction encodings. Simple transformations were used to create semantically equivalent instruction sequences that match some of the common instructions likely to be promoted to the IRF. Register re-assignment increased the instances where the register operands matched instructions in the IRF with the same opcode. Intra-block instruction scheduling improved the ability to place IRF instructions into loosely and tightly packed instructions by increasing the number of consecutive IRF instructions in a basic block. Inter-block instruction scheduling further improved the ability to pack IRF entries into the same instruction by moving instructions from successor blocks. Each of these transformations increased the pack density of an application, reducing the number of instructions that need to be fetched from and/or stored in the larger, less efficient instruction cache. Some transformations are unique to the IRF. For instance code duplication can be performed to move instructions to packed instructions in multiple preceding blocks to decrease code size and reduce power consumption. A unique form of predication is also possible by packing instructions after a conditional branch in a packed instruction. Energy savings for adding an IRF was 12.2% using an optimizing compiler targeting the IRF. The energy savings was increased to over 15.8% after the enhanced transformations were performed. Even considering that the original compiler [12] performed instruction selection, the energy savings by developing more targeted optimizations were further enhanced by approximately 1.07% which translates into a 7.2% improvement over the prior results. Static code size reduction was improved from 16.80% to over 28.83% after applying each of the adapted transformations. Again considering the previous instruction selection, this results in a net further 12.03% reduction, which is a 71.56% improvement over the existing results.

We believe that this paper shows how rethinking conventional optimization heuristics for our embedded architecture can lead to significant improvements in code quality. We further believe that compilers and their associated optimizations used to target embedded architectures with idiosyncratic ISA designs must be flexible enough to be easily modified in order to extract the full benefits of the underlying hardware.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants EIA-0072043, CCR-0208892, CCR-0312493, CCF-0444207, and CNS-0615085.

11. 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] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual*

International Symposium on Computer Architecture, pages 83–94, New York, NY, USA, 2000. 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. L. Corliss, E. C. Lewis, and A. Roth. A DISE implementation of dynamic code decompression. In *Proceedings of the 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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] S. Hines, G. Tyson, and D. Whalley. Improving the energy and execution efficiency of a small instruction cache by using an instruction register file. In *Proceedings of the 2nd Watson Conference on Interaction between Architecture, Circuits, and Compilers*, pages 160–169, September 2005.

[12] S. Hines, G. Tyson, and D. Whalley. Reducing instruction fetch cost by packing instructions into register windows. In *Proceedings of the 38th annual ACM/IEEE International Symposium on Microarchitecture*, pages 19–29. IEEE Computer Society, November 2005.

[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] 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.

[15] 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.

[16] Montanaro J., et al. A 160-mhz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Tech. J.*, 9(1):49–62, 1997.

[17] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[18] 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.

[19] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

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

[21] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 170–179, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

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

[23] 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.