Cool Code Compression for Hot RISC

Mark Hampton and Michael Zhang
MIT Laboratory for Computer Science, Cambridge, MA 02139
{mhampton|rzhang}@lcs.mit.edu

Abstract

Program size has become a very important constraint for embedded applications. There have been numerous efforts to reduce static code size, ranging from instruction set redesign to compiler compression techniques. One method, operand factorization, separates expression trees into tree-patterns and opcode-patterns and compresses them separately. This paper focuses on this technique and suggests extensions that can improve the compression ratio. We observe an average compression ratio of 61-63% for MediaBench and SPECint95 programs.

1 Introduction

Approximately half of the processor market is for embedded processors, which typically require low cost, small die area, and little power consumption. Because of the high cost and area requirements of memory, static code size has become a very important constraint for embedded applications. Although microprocessors employing RISC instruction sets benefit from simplicity in decoding logic, the low code density of RISC architectures causes program size to increase and also limits instruction cache bandwidth. High performance systems can also benefit from smaller code size due to the reduction in instruction cache miss rate. There are a variety of code compression algorithms that are employed to reduce code size. The metric used to compare compression algorithms is the ratio of compressed code size to original code size, also known as compression ratio.

Many techniques focus on ways to improve on the use of a standard compression algorithm, such as Huffman encoding, on a program. One such technique is operand factorization. This approach first divides a program into expression trees. The operand factorization technique then separates the operands from the expression tree, encodes the tree-patterns and operand-patterns separately, and recombines them in the compressed code. Based on studies conducted in [1], on average 20% of the tree-patterns cover almost all (over 95%) of the expression trees in SPECint95 programs. However, only about 80% of all the operand sequences are covered by 20% of the operand-patterns. In the operand factorization approach, a MIPS R2000 was used as the basis for investigation. In that architecture, the opcode bits can account for at most 35% of an instruction, meaning that the operand bits are more important in achieving a good compression ratio.

This paper proposes extensions to the operand factorization technique that are aimed at improving the compression of operand-patterns. First, we examine the effect of removing the requirement that the first instruction in a basic block must be the root of an expression tree. Next, we attempt to use the temporary nature of register values to reduce the variety of operands used in a program by allocating short-lifetime values to a single reserved register. We also propose additional techniques–trying to compress the code to save and restore registers for a procedure call more efficiently, and using an entire basic block as a unit of compression.

This paper is divided as follows. Section 2 describes related work in the area of code compression. Section 3 presents our methodology. Section 4 shows our results from implementing the baseline operand factorization approach. Section 5 describes the changes caused by removing the restriction that the first instruction in a basic block must be the root of an expression tree. Section 6 investigates the special handling of temporary values. Section 7 details additional proposed techniques. Section 8 concludes.

2 Related Work

A wide variety of approaches have been taken in the field of code compression. These techniques include instruction set redesign, various compiler techniques, and hardware decoding schemes. Many techniques build on the traditional methods of file compression, which can be divided into *statistical* and *dictionary*. Statistical compression looks at the entire program and replaces more frequently appearing text patterns with shorter codewords. A good example is the Huffman encoding algorithm. Dictionary compression, on the other hand, uses fixed length

codewords as indexes into the dictionary table. Statistical compression achieves a better compression ratio, but dictionary compression results in simpler and faster decoding.

ARM Thumb [5] and MIPS16 [7] try to use the dual-mode instruction sets which use a control bit to select the current instruction set for the decode logic, thus retaining the advantage of the wide but fast instructions and at the same time making it possible to use short instructions to condense code. To take it one step further, Larin et. al. [8] use the compiler to generate both a compressed program as well as fetch and decode logic specifically for the compressed code, with the assumption that most decoders are implemented with a PLA.

Various compiler techniques are also used to compress code size. In [9], common sequences are extracted to form a library and replaced with a procedure call. Suffixes of a larger common sequence can also be replaced with a procedure call and a starting point. Instead of considering each instruction as the smallest unit of compression, Araujo et. al. [1] breaks the instruction into two fields, opcode and operands, which achieves a better compression ratio. In [3, 13], it is observed that two equivalent intermediate code blocks could map to two different sequences of instructions, e.g., different register usages. Thus equivalence is determined on the control flow graph instead of final code sequences.

Another kind of compression technique involves hardware decoders which decode compressed instruction at run-time. The compressed code RISC Processor (CCRP) [14] is such an example. No hardware or instruction set modifications are necessary. However, this method does not improve instruction fetch bandwidth since decoding is done on the cache side. In [12], RISC instructions are compressed and hardware is used to gate the word lines to save fetch energy. This compression technique could also lead to code size reduction with small hardware modification.

3 Methodology

We generated code for a MIPS-II compatible processor using version 1.0.3a of the egcs compiler and version 2.8.1 of the gas assembler. All programs were compiled with the optimization flag -O2, and statically linked to produce an ELF executable. The generated code did not contain misaligned load/store instructions (LWL, LWR, SWL, SWR), synchronization instructions (LL, SC), or trap instructions (TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, TNEI).

For our workload, we use a combination of MediaBench and SPECint95 benchmarks, described in Table 1. Because the processor that we examined in our study does not contain a floating-point unit and emulates floating-point operations, we did not look at floating-point intensive code.

Benchmark	Description
adpcm (rawcaudio)	A speech compression program
g721 (decode)	A voice decompression program
gcc	A C compiler
go	A game-playing program
ijpeg	An image compression program
li	A Lisp interpreter
m88ksim	A microprocessor simulator
mpeg (mpeg2decode)	A MPEG decoder
vortex	An object-oriented database

Table 1: MediaBench and SPECint95 programs used in our study

As stated previously, we use compression ratio as our key metric, which is defined as the ratio of the size of the compressed program to the size of the uncompressed program. Due to time constraints, we were unable to generate compression ratio statistics for a variable-length encoding algorithm. Thus, we used a fixed-length compression algorithm, which results in a worse compression ratio than a variable-length algorithm but can still provide insight into how well our techniques work.

4 Baseline Implementation

We first implement the operand factorization technique as described in [1]. In this approach, an expression tree is used as the unit of compression. As defined in [2], which also uses an expression tree as the unit of compression, an instruction is the root of an expression tree if: (a) the instruction is a store operation; (b) the instruction computes a value used by more than one instruction in the basic block; (c) the instruction computes a value used outside the basic block; (d) the instruction is the first instruction in the basic block; (e) the instruction is a branch. For example, Figure 1 shows a sample code sequence that would form one expression tree, where the value computed by the `andi` instruction is only used by the `or` instruction, and the value computed by the `or` instruction is used outside of the basic block.

Using operand factorization, the operands of the expression tree are separated from the opcodes, forming an operand-pattern and a tree-pattern, respectively. These patterns are then compressed separately before being recombined. For the above expression tree, the tree-pattern

$$\begin{array}{rl} andi & \$r4, \$r4, 0xffffff00 \\ or & \$r3, \$r4, \$r1 \end{array}$$

Figure 1: An Code Sequence That Makes Up an Expression Tree

would be

$$\begin{array}{rl} andi & *, *, * \\ or & *, *, * \end{array}$$

and the operand-pattern would be

$$r4, r4, 0xffffff00, r3, r4, r1$$

Our implementation of the operand factorization technique relies on information generated by the compiler, assembler, and additional programs that we use on the binary executable. First, we use the static liveness information that is already maintained by the compiler. When the compiler determines that a value read by an instruction is being referenced for the last time–i.e. the value will be dead after the instruction executes–it appends a "`.l`" suffix to the assembly opcode with a corresponding operand number to indicate the last use of the value. For example, in the above instruction sequence, if the value in `r4` will not be referenced again after the or instruction, the compiler will output `or.l1`. If the value in `r1` will not be referenced again after the `or` instruction, the compiler will output `or.l2`. If neither value will be referenced again, the compiler will output `or.l12`.

This liveness information is then used by the assembler. When the assembler detects an instruction that computes a value used only by the subsequent instruction, it replaces the opcode of the first instruction with an unused opcode from the MIPS instruction set to indicate that the instruction is not the root of an expression tree. After the executable is generated, we then disassemble it using the `objdump` program, and parse the resulting output so that only the actual instructions (the numeric representation, not the assembler mnemonics or instruction addresses) are processed. We then pass these instructions to a program that uses the liveness information contained within the opcodes along with its own information about stores, branches, and basic block boundaries to subdivide the code into expression trees, and factor out the operands.

We present the results of our baseline implementation in Table 2. Our results are on the same order of magnitude as those generated in [1]; variations can be attributed to the fact that different toolsets were used and somewhat different versions of the MIPS architecture were targeted.

Program	Expression Trees	Tree-Patterns (%)	Operand-Patterns (%)
adpcm	7395	174 (2.35)	2546 (34.43)
g721	9120	217 (2.38)	3013 (33.04)
gcc	246214	1016 (0.41)	43049 (17.48)
go	59072	528 (0.89)	15149 (25.64)
ijpeg	43159	456 (1.06)	11077 (25.67)
li	22341	265 (1.19)	5222 (23.37)
m88ksim	32889	424 (1.29)	9008 (27.39)
mpeg	18160	332 (1.83)	5691 (31.34)
vortex	115909	409 (0.35)	16470 (14.21)

Table 2: Tree-pattern and operand-pattern occurrences in a program for our baseline implementation. The numbers in parentheses are the percentages of the total number of expression trees.

We also generated statistics on the cumulative percentages of expression trees covered by tree-patterns and operand-patterns, shown in Figures 2 and 3.

Using fixed-length encoding, we achieved an average compression ratio of 63.6%. The compression ratio statistics are shown in Figure 4.

5 Removing Restrictions on Heads of Basic Blocks

We removed the constraint that the first instruction in a basic block must be the root of an expression tree to determine if that would improve our results. As shown in Table 3, although the number of expression trees in the program decreases–which is expected since the trees can now include more instructions–the number of different tree-patterns and operand-patterns increases due to the fact that the additional instruction in the expression tree adds more variety to the possible patterns.

Statistics on the cumulative percentages of expression trees covered by tree-patterns and operand patterns are shown in Figures 5 and 6.

We discovered that removing the restriction that the first instruction in a basic block must be the root of an expression tree has a net positive effect. The average compression ratio with this constraint lifted improved to 60.6%, as shown in Figure 7. This indicates that there is room for improvement in the search for an ideal unit of compression.

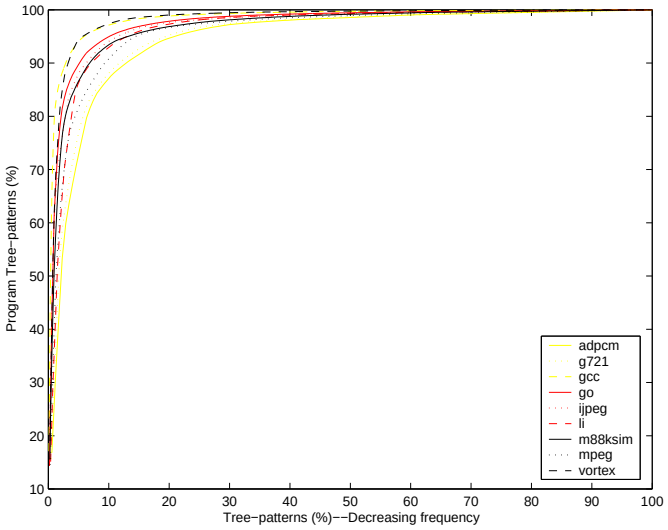

Figure 2: Percentage of covered tree-patterns for baseline implementation

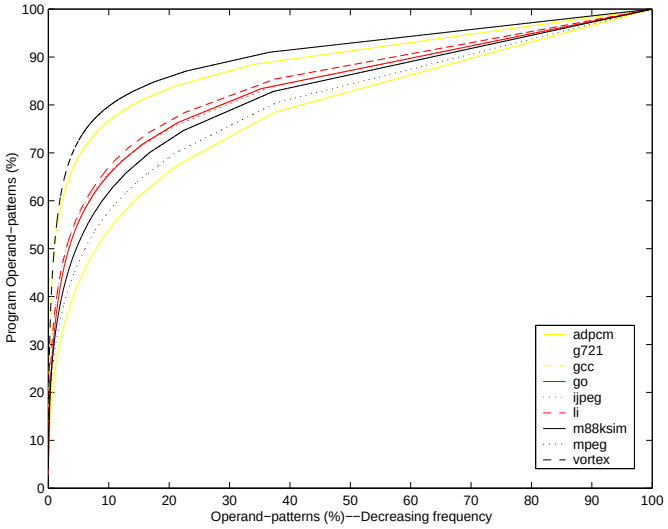

Figure 3: Percentage of covered operand-patterns for baseline implementation

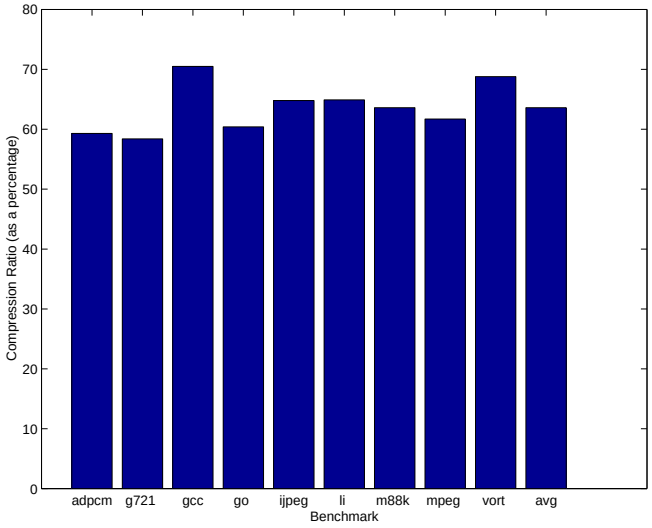

Figure 4: Compression ratio for baseline implementation

Program	Expression Trees	Tree-Patterns (%)	Operand-Patterns (%)
adpcm	6867	229 (3.33)	2599 (37.85)
g721	8576	282 (3.29)	3094 (36.08)
gcc	230446	1140 (0.49)	45642 (19.81)
go	54920	577 (1.05)	15209 (27.69)
ijpeg	41274	532 (1.29)	11529 (27.93)
li	21010	339 (1.61)	5451 (25.94)
m88ksim	30100	514 (1.71)	9371 (31.13)
mpeg	16776	402 (2.40)	5800 (34.57)
vortex	108851	484 (0.44)	17481 (16.06)

Table 3: Tree-pattern and operand-pattern occurrences in a program with the restriction removed that the first instruction in a basic block must be the root of an expression tree. The numbers in parentheses are the percentages of the total number of expression trees.

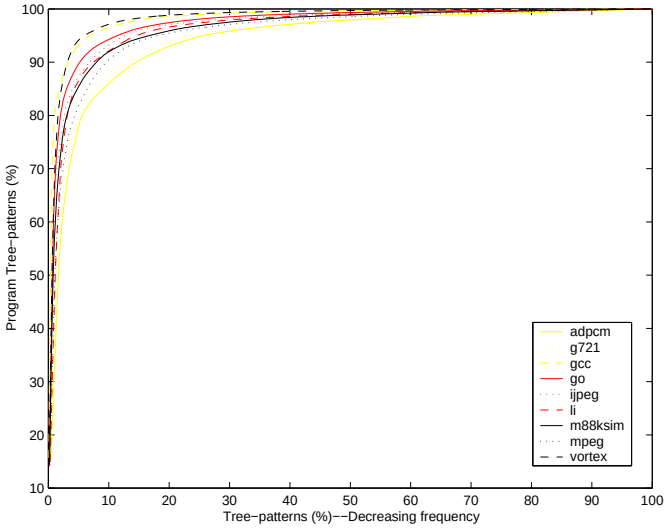

Figure 5: Percentage of covered tree-patterns with restriction on head of basic block removed

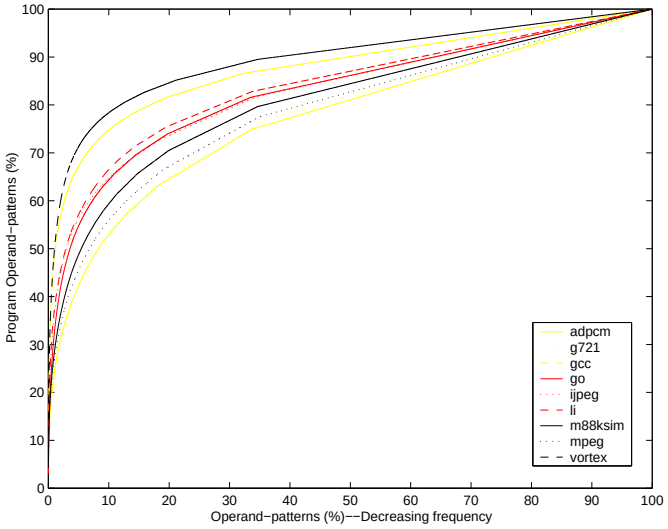

Figure 6: Percentage of covered operand-patterns with restriction on head of basic block removed

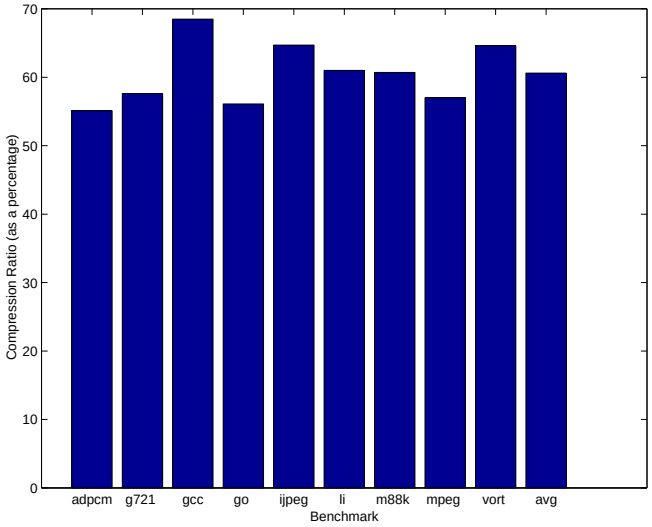

Figure 7: Compression ratio with restriction on head of basic block removed

6 Special Treatment of Temporary Values

It has been shown that many register values are short-lived, being referenced for the last time within a few cycles after it is produced [4], [10], [11], [6]. In fact, many instructions compute a value that is used only once, often by the immediately following instruction. For example, assembler macros often result in instruction pairs such as slt/bne or lui/lw, where the first instruction produces a value only used by the second. We investigated whether we could exploit this short-lifetime behavior to reduce the number of different operand-patterns in a program. For every instruction that computed a value that was only used once by the subsequent instruction and then never used again, we modified the assembler so that it would generate code to write that value to a single general-purpose register, which we reserved so that it would not be used in the compiler's register allocation. We observed that reserving a single general-purpose register had little effect on program behavior. Our results are presented in Table 4 and Figures 9, 10, and 11. As can be seen, the use of a single register to hold temporary values had an insignificant impact on the number of operand-patterns in the program and on the compression ratio. This can be attributed to the fact that although temporary values are stored in a single register, other values within the expression tree remain in their original registers, which means that there will still be considerable variety among the operand-patterns. However, this does not negate the merit of giving temporary values special treatment. An encoding could be devised that avoids having to encode the register for the temporary

Program	Expression Trees	Tree-Patterns (%)	Operand-Patterns (%)
adpcm	7400	174 (2.35)	2545 (34.39)
g721	9120	217 (2.38)	3013 (33.04)
gcc	246231	1016 (0.41)	42272 (17.37)
go	59093	528 (0.89)	15055 (25.48)
ijpeg	43232	458 (1.06)	11047 (25.55)
li	22341	265 (1.19)	5215 (23.34)
m88ksim	32888	424 (1.29)	8976 (27.29)
mpeg	18168	332 (1.83)	5677 (31.25)
vortex	115908	409 (0.35)	16410 (14.16)

Table 4: Tree-pattern and operand-pattern occurrences in a program where temporary values are stored in a single register. The numbers in parentheses are the percentages of the total number of expression trees.

value at all, thus reducing the number of bits required in the compressed instructions. To illustrate, consider the following code sequence:

```
addiu $r4,$r4,1
sw $r1, 0($r4)
```

If the value in r4 is not used again after the `sw` instruction, then we can generate the expression tree shown in Figure 8.

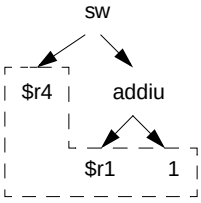

Figure 8: Modified Expression Tree

We feel this should improve the compression ratio to some extent, and that further investigation is warranted.

7 Additional Extensions

There are additional extensions worth exploring that we did not have time to implement. One extension attempts to improve the compression of the code generated by the compiler to save and store registers each time a procedure is called. For example, the following code segment could be part of a register-save sequence, where $sp refers to the stack pointer register.

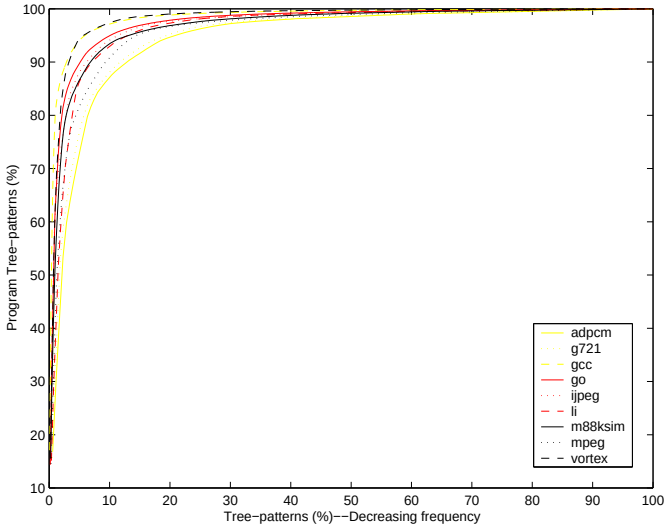

Figure 9: Percentage of covered tree-patterns with special handling of temporaries

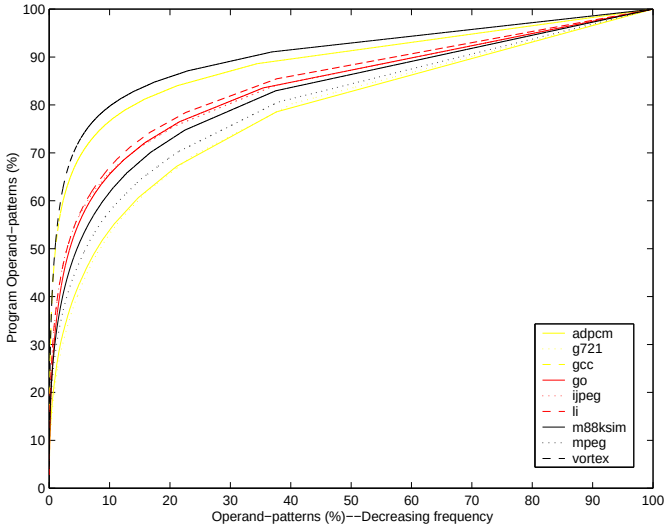

Figure 10: Percentage of covered operand-patterns with special handling of temporaries

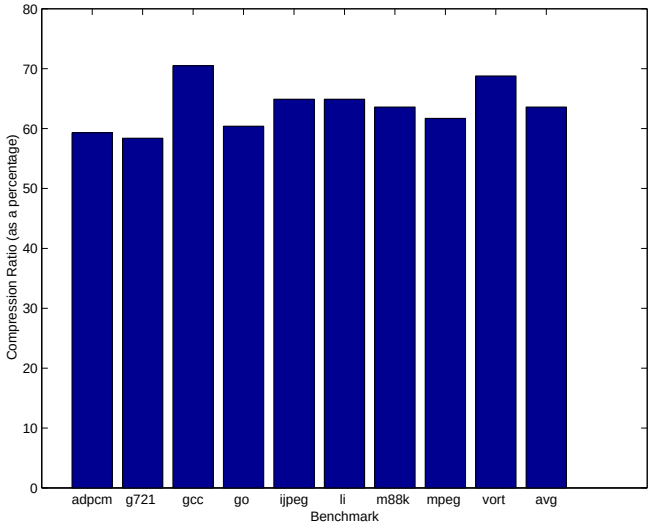

Figure 11: Compression ratio with special handling of temporaries

```
sw $r4, 12($sp)
sw $r3, 8($sp)
sw $r2, 4($sp)
sw $r1, 0($sp)
```

Instead of having the restriction that a store instruction must be the root of an expression tree, we can allow multiple stores to be encoded together, perhaps improving the compression ratio. A similar approach would apply to the load sequence used to restore registers before returning from a procedure call, an example of which is shown below.

```
lw $r4, 12($sp)
lw $r3, 8($sp)
lw $r2, 4($sp)
lw $r1, 0($sp)
```

These sequences can occur frequently in code, encoding them more compactly can potentially provide a big gain.

Another extension examines the effect of using an entire basic block as a unit of compression. Although this would probably result in a worse compression ratio due to the wide variety among basic blocks, it could provide interesting insight into the process of code generation.

8 Conclusion

This paper revisits the operand factorization code compression technique. It generates statistics for a baseline implementation of operand factorization and also implements two extensions: the removal of the restriction that the first instruction in a basic block must be the root of an expression tree, and the use of a single register to hold temporary values. Using a fixed-length encoding algorithm, the first extension improved the compression ratio by 3%. However, the second extension had an insignificant effect on compression. We feel that there is still an opportunity for improvement with respect to temporary values as an encoding can be developed that does not require the registers holding the values to be explicitly encoded. Two additional extensions are proposed as well: developing a more compact encoding of the code to save and restore registers for a procedure call, and determining the effect of using a basic block as the unit of compression.

We do not consider hardware decompression requirements in this paper. A possible area for future work is the examination of the area required by the decompression engine, as well as power consumption. In addition, different compression algorithms can be considered. Further improvement may be gained by reordering instructions within the assembler to produce more common sequences instead of accepting the compiler-scheduled code without modification. However, we do not feel that this improvement will be very great because many common code sequences are already generated by assembler macros or compiler algorithms that output code in a deterministic manner. But the effects of rescheduling code are still worth investigating.

References

[1] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression based on operand factorization. In *MICRO-31*, Dallas, Texas, December 1998.

[2] P. Centoducatte, R. Pannain, and G. Araujo. Compressed code execution on DSP architectures. In *isss12*, San Jose, California, November 1999.

[3] S. Debray, W. Evans, and R. Muth. Compiler techniques for code compression. Technical report, University of Arizona, Tucson, AZ, April 1999.

[4] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, December 1992.

[5] L. Goudge and S. Segars. Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications. In *Fourty-First IEEE Computer Society International Conference*, 1996.

[6] Z. Hu and M. Martonosi. Reducing register file power consumption by exploiting value lifetime characteristics. In *Workshop on Complexity-Effective Design, 27th ISCA*, Vancouver, Canada, June 2000.

[7] K. Kissell. MIPS16: High-density MIPS for the embedded market, 1997.

[8] S. Larin and T. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. In *Micro-32*, Haifa, Israel, November 1999.

[9] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *ARVLSI-95*, Chapel Hill, NC, March 1995.

[10] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292–302, December 1995.

[11] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 125–135, December 1997.

[12] M. Panich. Reducing instruction cache energy using gated wordlines. Master's thesis, Massachusetts Institute of Technology, August 1999.

[13] Johan Runeson. Code compression through procedural abstraction before register allocation. Master's thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, March 2000.

[14] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *MICRO-25*, December 1992.