Code Compression for DSP

Charles Lefurgy and Trevor Mudge
{lefurgy,tnm}@eecs.umich.edu
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122
http://www.eecs.umich.edu/~tnm/compress

Abstract

Previous works have proposed adding compression techniques to a variety of architectural styles to reduce instruction memory requirements. It is not immediately clear how these results apply to DSP architectures. DSP instructions are longer and have potentially greater variation which can decrease compression ratio. Our results demonstrate that DSP programs do provide sufficient repetition for compression algorithms. We propose a compression method and apply it to SHARC, a popular DSP architecture. Even using a very simple compression algorithm, it is possible to halve the size of the instruction memory requirements.

1 Introduction

DSP architectures have adopted several characteristics of VLIW, including wide instruction words. The cost of using the explicit parallelism of VLIW is much larger code sizes. Beyond the classical optimizations used to achieve smaller programs, compression can shrink program size by utilizing repetition found at the instruction level. Several compression techniques have been proposed for general purpose architectures [Wolfe92, Kozuch94, Liao95, Benes97, Ernst97, Kirovski97, Lefurgy97, Wolf97, Aranjo98]. Previous work focused on using short variable-length codewords and increasing the meaning of codes by allowing them to decode to a list of instructions. It is not known if such compression methods can be used on DSP architectures. DSP instructions can hold multiple independent operations which potentially increases variance in the instruction bit patterns. Our previous study [Lefurgy98] noted that most compression can be attributed to single instruction patterns. We use this idea to show that programs for DSP architectures are highly compressible. Compression for DSP has two important ramifications. First, performance can be traded for small code size. Second, small code size reduces the frequency at which overlays are performed and therefore can vastly improve execution time.

The organization of this paper is as follows. Section 2 reviews previous work in code compression. We present our compression method in section 3. Our experimental results are presented in section 4. In section 5, we discuss some implications of the results. Finally, section 6 contains our conclusions.

2 Previous work

There have been several recent works on code compression. The Compressed Code RISC Processor (CCRP) [Wolfe92, Kozuch94, Benes97] is a MIPS processor that compresses instruction cache lines using Huffman coding. Dictionary compression methods [Bell90] have been studied for several processors [Liao95, Lefurgy97]. A software-managed compression-cache that decompresses functions on a cache miss has been proposed [Kirovski97]. Compression algorithms based on operand factorization and Markov models have been suggested for transmitting programs over networks [Ernst97]. Carmel [Sucher98] is a DSP architecture that uses a dictionary compression technique. More complicated compression algorithms have combined operand factorization with Huffman and arithmetic coding [Wolf97, Aranjo98]. A VLIW program representation [Conte95] reduced program size by eliminating NOP fields.

In a previous work [Lefurgy97], we used dictionary compression to reduce the instruction memory footprint of embedded programs. We examined replacing frequently used sequences of instructions with a codeword. The codeword served as an index into a list of instruction sequences. Fetching and decoding the codewords recovered the original sequence of instructions to execute. A variable-length encoding using small codewords (8-bits, 12-bits, and 16-bits), allowed us to compress PowerPC programs to 60% of their original size. We will show that even simpler compression techniques can improve SHARC [ADI] programs by much greater amounts.

3 Compression architecture

Our compression scheme takes advantage of the observation that the instructions in programs are highly

October 1, 1998
Submitted to CASES-98

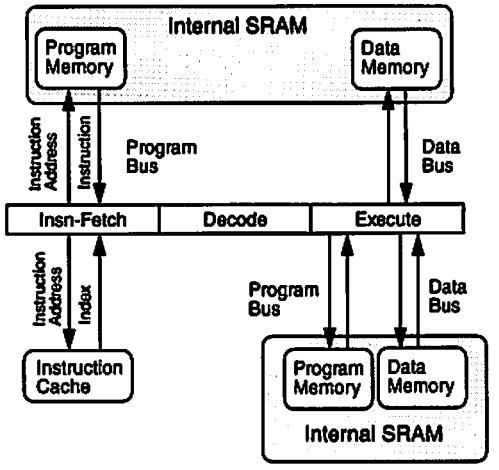

Figure 1: SHARC pipeline

Top shows program fetch during execution of a single data access instruction. Bottom shows program fetch during execution of a dual data access instruction. Instructions are fetched from cache when execution unit uses both Program and Data busses to fetch data.

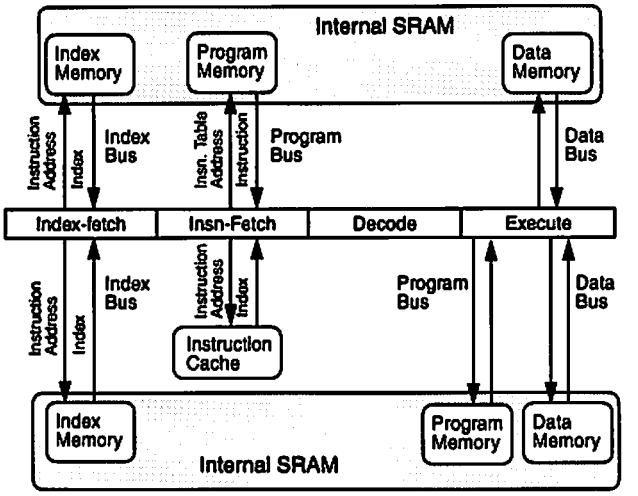

Figure 2: Compressed program pipeline

Top shows program fetch during single data path instruction execution. Bottom shows program fetch during dual data path execution.

repetitive. Each unique instruction word in the program is put in an *instruction table*. Each instruction in the program is then replaced with an index into this table. Because the instruction words are replaced with a shorter code and because the table overhead is usually small compared to the program size, the compressed version is smaller than the original. Instructions that only appear once in the program are problematic. The original instruction in the instruction table and the index in the program stream are larger than the single original instruction, causing a slight expansion from the native representation.

The SHARC pipeline is shown in Figure 1. SHARC typically uses the Program Memory bus to fetch instructions and uses the Data Memory bus to fetch data. However, it can also use these busses for dual data access. When this happens, instructions are executed from the instruction cache so that the Program Memory bus can be used for data fetch. The modification of SHARC for compressed programs is given in Figure 2. We augment the 3 stage SHARC pipeline by adding a pre-fetch stage. First, the pre-fetch stage retrieves the 16-bit instruction index from the external memory. The *instruction table address register* holds the location of the instruction table in the internal memory. Adding the contents of this register to the index forms the address of the SHARC instruction. Second, the fetch stage uses this address to get the 48-bit SHARC instruction word. Finally, the instruction is issued to the decode stage.

There are three costs for adding the pre-fetch stage. First, an extra internal memory bus must be added to support simultaneous access to the index memory, program memory, and data memories. SHARC uses dual-ported SRAM to achieve simultaneous accesses over the program and data busses. Instead of adding another port to SRAM for the index bus, a separate SRAM block could be dedicated to index memory. Second, the pre-fetch stage adds a third branch delay slot. Last, one register must be added to hold the address of the instruction table.

When data and program accesses compete for use of the program bus, SHARC puts the conflicting instruction in the instruction cache. Future references to the same instruction address can use the I-cache and allow the program bus to be used for data. This feature allows loops with instructions that use the program bus for data access to execute without penalty due to bus contention. This is extremely important for DSP algorithms which tend to be composed of small, computation-intensive loops. Our compression architecture retains this valuable feature.

The 16-bit index limits a program to use only 64K unique instructions. However, programs that use more instructions can be accommodated. One alternative is to add a mode-switching branch to the instruction set similar to the one used in ARM [ARM95, Turley95]. This would cause the fetch units to switch between using indexes and normal SHARC instructions. In native mode, the pre-fetch stage could be turned off. The fetch stage would use the program counter to fetch SHARC instructions as usual. Another possibility is to encode different parts of the program by using different instruction tables. By simply reloading the *instruction table address register*, an entire

Benchmark	Optimization	Static Instructions	Table size	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio
mpeg2enc	none	28,832	7,167	172,992	100,666	58.2%
	-O1	26,537	8,118	159,222	101,782	64.0%
go	none	81,343	8,564	488,058	214,070	43.9%
	-O1	78,459	13,413	470,754	237,396	50.4%
ghostscript	none	352,525	33,322	2,115,150	904,982	42.8%
	-O1	310,869	49,734	1,865,214	920,142	49.3%

Table 1: Baseline results

new set of 64K instructions can be used. This register can also be used to allow each program in an embedded system to use its own instruction table so that the tables are tuned to the instructions that the program actually uses.

3.1 Branch instructions

In our previous work, we did not compress branch instructions because doing so could affect instruction repetition in complicated ways. Using patterns of only 1 instruction with a fixed-length encoding eliminates this problem. Compressing a program moves all instructions to a different location. This affects branches which have index and address fields. Additionally, codewords are smaller than the original instructions, so the instruction fetch mechanism and branches must use a new alignment. Since we are using 16-bit codewords, PC-relative branches and absolute branches now specify a 16-bit aligned address. In this simple scheme, the index fields of the PC-relative branches do not change since the distance (number of instructions or codewords) between the branch and target are the same. Absolute branches, which the compiler uses for function calls, must change to use the address for the new location of the target function. However, all such branches that matched before will also match after this transformation. Therefore, we can easily compress branch instructions just as any other instruction.

4 Results

In this section we integrate our compression technique into the SHARC ADSP-2106x instruction set. We use benchmarks from SPEC CINT95 [SPEC95] and MediaBench [Lee97]. These benchmarks are compiled with the VisualDSP compiler from Analog Devices. The portions of the benchmarks for file I/O were removed since they are not supported by the compiler's libraries. Our results include both application and library code. All compressed program sizes include the overhead of the dictionary. *Compression ratio* is used to measure the amount of compressibility.

$$compression\ ratio = \frac{compressed\ size}{original\ size} \qquad (Eq.\ 1)$$

Table 1 shows the results for the our basic compression method. Each benchmark was compiled with and without optimizations. We only use "-O1" optimization because higher levels of optimization exposed bugs in the compiler. The *Table Size* column is the number of entries in the *instruction table*. There is one entry for each unique instruction bit pattern in the program. *Compressed Size* is the combined size of the indexes and the instruction table.

Classical code optimizations are one way to attain a smaller code size. Using some optimization on the benchmarks reduces the number of instructions, but it also increases the table size. The table size increases because the number of unique instructions increases when single operation instructions are combined into 2 and 3 operation instructions. In un-optimized code, instructions only contain 1 operation and are more likely to match each other. The reduced number of instructions in the optimized code did not account for the increase in the table size. Therefore, the smallest representation was attained by compressing un-optimized code.

The instruction tables contain many instructions that are used only once in the entire program. One reason this happens is that the combination of registers the register allocation algorithm uses for a particular instruction may not match any other instruction. We can improve the compression ratios by removing these unique instructions from the table. To accomplish this, we select some instructions that can be represented in 16-bits and mix them in with the index stream. These short instructions will be coded with unused index values. For this experiment, we selected the 8 most frequent ALU operations for each benchmark to use as short instructions. The encoding of the index stream is as follows. If an index begins with the bit 0, then the remaining 15 bits are the index into the instruction table. If then index begins with 1, then next 3 bits will select an ALU operation in the SHARC. The remaining 12 bits are divided into groups of 4-bits to select 3 registers for the ALU operation. The 3-bit ALU operation field selects one

Benchmark	Optimization	Table size	Table size change from baseline	Compressed size (bytes)	Compression ratio	Compression ratio change from baseline
mpeg2enc	none	6,107	-1060	94,306	54.5%	-3.7%
	-O1	7,323	-795	97,012	60.9%	-3.1%
go	none	7,213	-1351	205,964	42.2%	-1.7%
	-O1	11,728	-1203	223,216	48.7%	-1.7%
ghostscript	none	29,183	-4139	880,148	41.6%	-1.2%
	-O1	46,498	-3236	N/A	N/A	N/A

Table 2: Addition of short instruction words

Benchmark	Optimization	Compressed size (bytes)	Compression ratio	Compression ratio change from baseline
mpeg2enc	none	89,647	41.9%	-16.3%
	-O1	88,541	44.6%	-19.4%
go	none	196,261	40.2%	-3.7%
	-O1	203,629	44.4%	-6.0%
ghostscript	none	883,783	41.8%	-1.0%
	-O1	852,865	45.7%	-3.6%

Table 3: Nibble encoding

entry from an 8-entry table of SHARC ALU opcodes. This table could be programmable so that each program could select the 8 best ALU instructions to help compression.

Results for mixing ALU operations and indexes are presented in Table 2. Some common ALU operations used are addition, multiplication, subtraction, pass, compare, increment, and decrement. This encoding significantly reduces the table size. However, for *mpeg2enc* with optimization, there are too few unused index values to add the shortened ALU instructions. For other benchmarks, the compression ratio is improved between 1.2% and 3.7%.

5 Discussion

For comparison, we also compressed the benchmarks with a nibble compression algorithm [Lefurgy97]. This algorithm reduces the size of codewords (indexes) to 8 bits, 12 bits, and 16 bits. Each codeword can represent a list of instructions. However branch instructions are not encoded. Instead they are prefixed with an 4-bit escape nibble to differentiate them from the codewords. Table 3 shows the results and compares them to the baseline method. This demonstrates that more complicated schemes can attain better compression ratios. Interestingly, the compression ratios for the larger benchmarks are quite similar which shows that even simple compression algorithms can be effective. Using the shorter codewords instead of compressing branches yielded slightly better compression ratios for the larger benchmarks.

In embedded systems that must use external memory to store programs, overlays are an important way to effectively use internal memory to achieve high performance. Code compression can assist such systems to achieve even greater performance. Smaller code size reduces the frequency at which overlays must be used since a larger portion of the program can fit in internal memory. In addition, loading a compressed function from external memory requires less time than loading a non-compressed function.

6 Conclusions

We have demonstrated that even simple compression methods can be highly effective at reducing code sizes in DSP programs. Compressing only single instructions to a fixed-length code allows us to have a simple mechanism for decompression which has minimal impact on the SHARC architecture. Our method can compress programs to half their original size while allowing the hand-coded numerical loops that are important in DSP algorithms to run at native speeds.

Acknowledgments

This work was supported by DARPA grant DABT63-97-C-0047.

References

[ADI] Analog Devices, Inc., *SHARC User's Manual.*

[ARM95] Advanced RISC Machines Ltd., *An Introduction to Thumb*, March 1995.

[Bell90] T. Bell, J. Cleary, I. Witten, *Text Compression*, Prentice Hall, 1990.

[Benes97] M. Benes, A. Wolfe, S. M. Nowick, "A High-Speed Asynchronous Decompression Circuit for Embedded Processors", *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[Conte95] T. Conte and S. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures", *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[Ernst97] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression", *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, June 1997.

[Fraser95] C. W. Fraser, T. A. Proebsting, *Custom Instruction Sets for Code Compression*, unpublished, http://www.cs.arizona.edu/people/todd/papers/pldi2.ps, October 1995.

[Kirovski97] D. Kirovski, J. Kin, and W. H. Mangione-Smith, "Procedure Based Program Compression", *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[Kozuch94] M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," *IEEE International Conference on Computer Design*, 1994.

[Lee97] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[Lefurgy97] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving code density using compression techniques", *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[Lekatsas98] H. Lekatsas and W. Wolf, "Code Compression for Embedded Systems", *Proceedings of the 35th Design Automation Conference*, June 1998.

[Liao95] S. Liao, S. Devadas, K. Keutzer, "Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques", *Proceedings of the 15th Conference on Advanced Research in VLSI*, March 1995.

[SPEC95] SPEC CPU'95, Technical Manual, August 1995.

[Sucher98] R. Sucher, R. Niggebaum, G. Fettweiss, and A. Rom, "CARMEL - A New High Performance DSP Core Using CLIW", *9th Annual International Conference on Signal Processing Applications and Technology*, September 1998.

[Turley95] J. L. Turley. "Thumb squeezes arm code size". *Microprocessor Report*, 9(4), 27 March 1995.

[Wolfe92] A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.