Technical Report CSE-TR-330-97, University of Michigan

The Impact of Instruction Compression on I-cache Performance

I-Cheng K. Chen
Peter L. Bird
Trevor Mudge

EECS Department
University of Michigan
{icheng,pbird,tnm}@eecs.umich.edu

Abstract

In this paper we present a straightforward technique for compressing the instruction stream for programs that overcomes some of the limitations of earlier proposals. After code generation, the instruction stream is analysed for frequently used sequences of instructions from within the program's basic blocks. These patterns of multiple instructions are then mapped into single byte opcodes. This constitutes a compression of multiple, multi-byte operations onto a single byte. When compressed opcodes are detected during the instruction fetch cycle of program execution, they are expanded within the CPU into the original (multi-cycle) sequence of instructions. We only examine patterns within a program's basic block, so branch instructions and their targets are unaffected by this technique allowing compression to be decoupled from compilation.

1. Introduction

Compilers are universally used for program development, due to the complexity of managing the large applications developed today. However, despite the sophistication of the optimization process, the code generated by compilers can be sub-optimal and wasteful of program space, in part because compilers expand common syntax program fragments into machine instructions using a common set of mapping templates.

Even today, hand tuned assembly code is developed for those program fragments which have been found, by profiling, to execute frequently. Since programs spend most of their time in a few sections of code (the "80/20" rule is a common rule of thumb), these hand-tuning for small sections of the program can have important performance payoffs.

In this paper, we investigate a compiler optimization to reduce program size and thereby improve the performance of a machine's instruction cache. We compress a program based upon finding redundant instruction sequences in the binary instruction stream and remapping these onto a single byte opcode.

1.1 Patterns

Compilers generate code using a *Syntax Directed Translation Scheme* (SDTS) [1]. Syntactic source code patterns are mapped onto templates of instructions which implement the appropriate semantics. Consider, a simple schema to translate a subset of integer arithmetic:

```
expr ->     expr'+' expr
    {   emit( add, $1, $1, $3 );
        $$ = $1;
    }

expr ->     expr'*' expr
    {   emit( mult, $1, $1, $3 );
        $$ = $1;
    }
```

In these patterns, the expression subtrees on the RHS of the productions return registers which is used by the arithmetic operation. The register number holding the result of the operation ($1) is passed up the parse tree for use in the parent operation. These two patterns would be reused for all arithmetic operations throughout all generate programs. The only difference in instruction sequences would be the register numbers used in the arithmetic operations.

More complex actions (such as translation of control structures) generate more instructions, albeit still driven by the template structure of the SDTS.

Although the code generation scheme outlined above is part of one method of translation (namely, pattern matching code generation), these reasoning applies to other techniques for instruction selection and code generator. In general, the only difference in instruction sequences for given source code fragments at different points in the object module are the register numbers in arithmetic instructions and operand offsets for *load* and *store* instructions. As a consequence, object modules are generated with many common sub-sequences of instructions. There is a high degree of redundancy in the encoding of a program.

Object Oriented Programming languages encourage the increase of redundancy in programs. *Information hiding* is one organizational strategy (among many) used for OOP. The implementation of an object is hidden within the private namespace of the class, with member functions used as the interface to the object. Often, these member functions are simple access routines which reference private member data structures. These short code sequences are also pattern templates, similar to the SDTS of a compiler.

1.2 Instruction Fetch Bottleneck

Although most work has been done in optimizing data fetch behavior, it is clear that instruction fetch is also a significant bottleneck for high performance computing. An I-cache miss will stall the processor, while the instruction memory access bandwidth limits the rate at which new instructions can be delivered to the processor.

A recent study [9] examined instruction fetch and observes that “code bloat” caused by the growth in application binaries, with each new version, and increasing use of operation system services is resulting in larger and larger load modules that put increasing pressure on instruction caches.

A study at DEC [6] showed a very similar result. When executing an SQL server on a DEC 21064 Alpha, the program is bandwidth limited by a factor of two on instruction cache misses alone. Moreover, the CPI is increased to 4.3, almost an order of magnitude reduction from its peak performance.

The instruction compression technique presented in this paper can effectively solve the above mentioned problems. Because this scheme reduces the size of programs, it effectively increases the effective size of an I-cache for the same program fragment (a code stream is dynamically expanded within the CPU rather than redundantly occupying precious cache lines). Moreover, since fewer bytes are transferred from program memory to the I-cache, the instruction memory bandwidth requirements of the program are reduced.

2. Description of the Compression Technique

Given the high degree of redundancy of instruction streams, an effective compression of redundant sequences should improve instruction fetch behavior. In this section, we describe our instruction compression technique.

After code generation and register allocation, we analyse the generated code stream to search for patterns. Patterns are built from the bit patterns of sequences of machine instructions. The pattern search is made over instruction sequences (fixed length 8 or 16 instructions) *within* all basic blocks of the program[1]. An incidence count is maintained for each unique pattern. The resulting set of patterns serves as the basis for program compression.

We have also annotated each instruction with its execution frequency established through profiling. This profile information is accumulated with each pattern usage occurrence in the program.

After the pattern set has been constructed, a simple tiling procedure is applied to the program. Those patterns with the highest frequency of usage are encoded as one byte, and the original sequence of instructions for the pattern is replaced by this opcode. Incidence counts for patterns which overlap the selected pattern are appropriately decrement.

1. A basic block starts at a label (or after a branch operation) and ends with a branch operation (or another label).

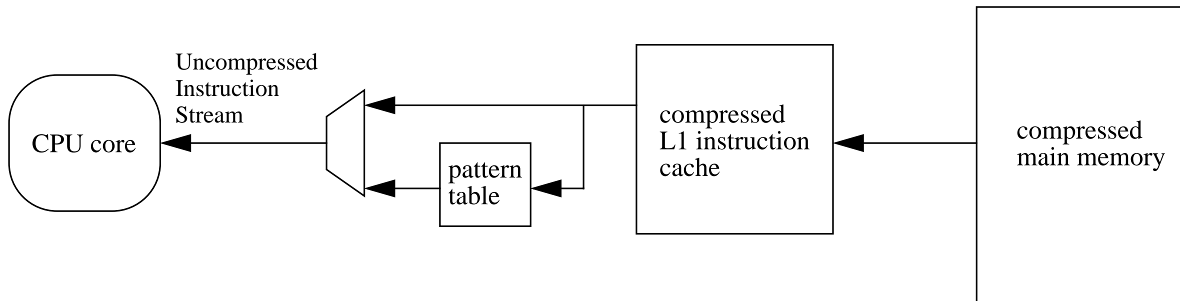

Figure 1. CPU Layout with Table for Expansion of Instructions

The original instruction sequence is saved in a table in the CPU (see Figure 1). During instruction fetch, the instruction decoder checks the opcode of the incoming instruction. If the opcode indicates an uncompressed instruction, then instruction decode and execution proceeds in a conventional fashion. When the decoder encounters a compressed instruction, the entire sequence of instructions is retrieved from the ROM and dispatched through the execution pipeline sequentially. Instruction fetch from the I-cache is avoided until the sequence completes.

We have incorporated the decode table as a ROM in our CPU model. However, if the cost of loading this table is not high (with respect to the execution of the associated thread), then this table could be part of the state of the process.

3. Experiment

In an earlier work [2], we examined the impact of compression on program size. In that work, we found that our instruction compression technique could reduce overall program size by 45% to 60%.

Since I-cache behavior (and instruction bandwidth requirements) are inherently dynamic properties of program behavior, we wanted to investigate instruction compression based upon program execution. To measure the impact of compression on dynamic behavior, we conducted trace-driven simulations. As input for the simulation, we use SPEC CINT95 and CFP95 benchmark suites [8].

We do not currently have a compiler that generates code to support this optimization. Therefore, we simply used the output generated by a existing compiler. All benchmarks were compiled with the DEC C compiler and DEC fortran compiler using -O optimization flag. Because code was not generated to take advantage of this peephole optimization, we feel that the performance information presented in this paper represent a “worst-case” optimization capability for this compression technique.

To collect profile information and instruction traces, we used ATOM [4], a code instrumentation interface from Digital Equipment Corporation. The benchmarks are first instrumented with ATOM, then executed on a DEC 21064-based workstation running the OSF/1 3.0 operating system to generate traces. These traces contain only user-level instructions.

The profile information for the programs was attached to each basic block, and patterns were constructed from resulting assembler listing. Since we assume that a pattern set can be viewed as a part of the process state, we used the same programs for “training” as for simulation.

The Instruction cache we simulated was a direct mapped implementation. The I-cache capacity measured was 2K through 32K. We measured caches with line sizes of 4 and 8 words (16 and 32 bytes). We looked at the traffic from the CPU to the first level cache.

Since we have compressed programs into a single byte opcode, we have made the necessary assumption that instruction words are not aligned.

The statistics of traces from the SPEC benchmarks are summarized in Table 1. The table shows the number of instructions and basic blocks of each program.

3.1 Results

In examining the results of our simulations, we were interested in two distinct machine behaviors:

1. The number of bytes fetched by the CPU from the Level 1 I-Cache.

Table 1. Instruction Counts

	Benchmark	Static Instructions	Basic blocks
Integer	compress	2260	223
	gcc	351936	29,054
	go	62380	8,877
	ijpeg	47988	2,161
	li	16752	1,167
	perl	87172	3,252
	vortex	142640	12,187
Floating point	applu	13064	534
	fppp	12848	334
	hydro	9528	1,018
	su2cor	11056	875
	swim	2064	178
	tomcatv	1308	128

2. The miss ratio for the I-cache for different cache sizes.

These two sets of values were collected and compared for both compressed and uncompressed instruction traces.

The byte fetch demands provides an index of the demands upon the I-cache from the CPU. By comparing compressed and uncompressed programs, we can get a measure of the reduction in requirements caused by compression.

The I-cache miss ratio is a measure of the relative effectiveness of use by compressed and uncompressed instruction streams.

3.1.1 I-cache Fetch Behavior

We compared the I-cache fetch requirements of programs with compressed instruction sequences against those with uncompressed streams. Figure 2 and Figure 3 show the results. Figure 2 shows the results of the *integer* programs from the SPEC benchmarks, while Figure 3 shows the *floating point* program set.

Each chart is organized to show the relative performance of each benchmark in its compressed implementation against the uncompressed version. The value of 100% indicates the fetch costs incurred by the uncompressed version of the program while the bars measure the compressed I-stream programs.

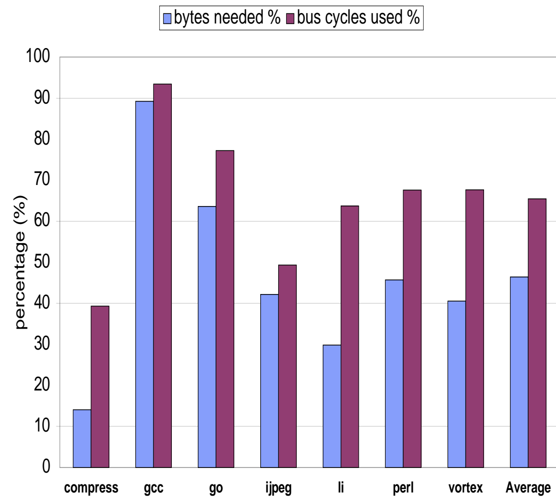

Figure 2. Bytes needed and bus cycles used for SPEC CINT95 benchmarks

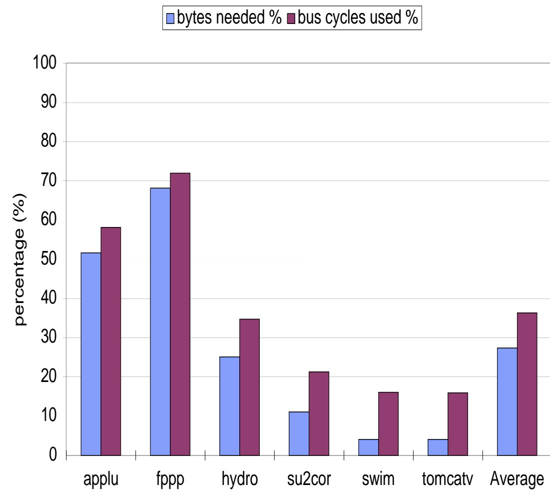

Figure 3. Bytes needed and bus cycles used for SPEC CFP95 benchmarks

Associated with each program are two cost bars. The left-most bar indicates the number of actual bytes required by the CPU for program execution. This, of course, is a measure of the absolute minimum number of bytes required. Since each I-cache fetch may result in multiple bytes delivered to the CPU (to pad a 4 byte instruction path from I-cache to CPU), this number may be too optimistic. Therefore, we have also measured the number of bus cycles required for instruction transfer; this is the right-most bar for each program. We have provided the two views mainly for comparison. Although it

appears that the *byte-fetch* measure may be too optimistic an assessment of I-cache fetch behavior, any machine that uses a pre-decode phase [5] or has queue to hold incoming instructions would more likely approximate the byte fetch behavior recorded with the first bar.

The last entry of both charts represent the average performance of all programs in the associated benchmark set (integer or floating point).

Looking at the integer programs, we can see that the number of bytes fetched by the CPU from the I-cache is, on average, less than 50%. Moreover, this average is highly skewed by the performance of the gcc compiler. Even assuming that each fetch from the I-cache requires a 4 byte transfer, on average we have reduced access to the first level cache by 35%.

Looking at the floating point benchmarks, the impact on I-cache traffic becomes more pronounced. Overall, the instruction fetch traffic to the I-cache is reduced by over 70%. If we assume a full, 4 byte transfer is required for each instruction fetch, we have still reduced the I-cache access requirements by 65%

3.1.2 I-cache Miss Behavior

The floating point benchmarks from SPEC are all very small. Their code segments are often small enough to fit entirely into even a small I-cache. This eliminates I-cache misses for all but the smallest I-cache configurations.

For this reason, we will concentrate upon examining the I-cache miss behavior of the integer benchmarks.

Consider the SPEC integer benchmarks. The average miss ratios for these benchmarks are shown in Figure 4, (the miss ratios for each individual benchmark are shown in the Appendix).

When programs are compressed with 128 patterns (that is, the128 most active instruction sequences are compressed to single opcodes), the average miss ratio is less than the miss ratio for an uncompressed program utilizing a cache with twice the capacity. Even when only 32 patterns are used, the overall I-cache rate is reduced. This configuration achieves about half the miss rate reduction shown with 128 patterns.

3.1.3 Pattern Table Sizes

As discussed in Section 2. above, all instructions of each basic block are into sets of patterns. The original sequence of instructions is saved in a table in the CPU which is accessed during instruction fetch and decode to retrieve the original sequence of instructions. Since the table occupies an expensive resource of the CPU, it is desirable that its size be small.

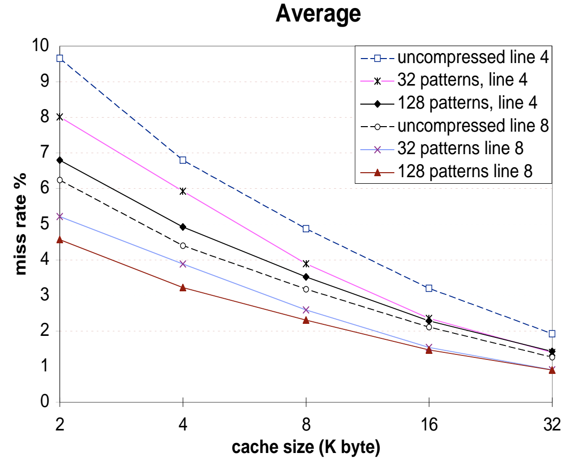

Figure 4. Average miss rates for SPEC CINT95 benchmarks

Table 2 shows the size of the remapping table which would be required for each application, given a fixed number of patterns were included in the remapping set.

Table 2. Pattern Table Sizes

	8	16	32	64	128
compress	228	448	836	1,400	1,776
gcc	172	348	780	1,332	2,208
go	156	340	648	1,248	2,372
ijpeg	196	392	820	1,792	3,776
li	204	348	564	824	1,132
perl	208	376	704	1,176	1,788
vortex	196	292	592	980	1,748

The table shows the set of integer benchmarks, and a list of possible pattern ranges. For all pattern sets less than 32, the remapping table size is less than 1K. When 128 patterns are used, the remapping table is in the range of 1K to 4K.

Figure 5 shows the impact of the increase in pattern counts upon the *byte-fetch* behavior. Shown are the performances for the integer benchmarks using 32 patterns and 128 patterns. For each benchmark, 4 bars are shown:

- 128 Patterns: Byte fetch requirements
- 128 Patterns: Byte fetch, padded to 4 bytes

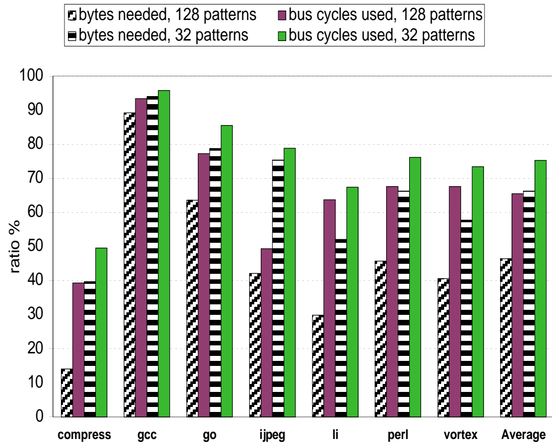

Figure 5. Comparison of the compression effects with 32 patterns and 128 patterns

- 32 Patterns: Byte fetch requirements
- 32 Patterns: Byte fetch, padded to 4 bytes

On average, the 32 pattern implementation reduced the I-cache fetch requirements by 35%.

4. Related Research

In [10], Wolfe and Chanin propose a compression scheme for embedded microprocessors. Their scheme uses the instruction cache to perform the instruction expansion. Since the cache resident version of the program is uncompressed, the miss rates and the bandwidth to the CPU are not improved. Their scheme also involves other overheads. Programs are compressed using a Huffman encoding which requires additional overhead in decompression. In addition, they require large address remapping tables and extra translation hardware to maintain the relationship between the compressed and the uncompressed address spaces.

An interesting contrast to our scheme is the Trace Cache [7]. The Trace Cache increases the effective bandwidth by combining information from multiple basic blocks. On the other hand, our scheme increases the effective bandwidth by condensing the information in each single basic block. The fundamental difference in the two schemes is that the Trace Cache captures information about program flow *dynamically*, while our compression technique is *static*; all compression is discovered during compilation. Thus, the two approaches are orthogonal. They might be applied in conjunction to achieve a greater benefit.

5. Conclusion and Future Research

The instruction stream represents a significant bottleneck to high performance execution. In this paper we have outlined a technique for reducing a program's instruction stream by compressing frequently encountered instruction sequences into single byte opcodes.

We have examined remapping of instruction streams for both static and dynamic models. When we applied the technique for static pattern incidences [2], we were able to reduce overall program size by 45% to 60%. Although our compression technique is adversely affected by heavy usage of large register sets (each different register will result in different bit patterns in instructions which are compared for patterns), we found that we could get effective compression both on CISC and RISC architectures.

In the study presented in this paper, we are interested in the impact of instruction stream compression on the I-cache behavior for a given architecture. We do not have a compiler that is organized for this optimization, so we used the output of an optimizing compiler for the Alpha RISC architecture.

Despite the fact that the compiler was not tuned to exploit instruction compression, we were able to reduce both the I-cache byte fetch requirements and the I-cache miss rates for the integer programs from the SPEC benchmarks.

We believe that the impact of instruction stream compression will be much higher when we incorporate this optimization technique more directly into the compiler. This technique could be integrated either by rebuilding the code generation phase (most specifically, register allocation), or by performing a peephole optimization pass over the generated instructions (possibly reassigning registers so as to increase the common pattern count and hence increase pattern incidences).

Since we are constructing patterns based solely upon the bit patterns of individual instructions, the varied use of registers by a compiler has a negative impact upon pattern incidence counts. And yet, although using only a very small register set size will increase pattern incidences, it will likely have an adverse effect on overall program execution. It is important, therefore, to identify strategies that both lead to high performance code sequences, and aid in the generation of compact code. At least two strategies might be employed.

It is possible for the compiler to establish machine "idioms" (much like a VECTOR opcode). The registers, instructions, and operand offsets for the operands to these "meta-operations" would need to be fixed.

Another possible implementation could use a smaller set of patterns (hence a smaller remapping table), but limit its scope to a subset of the executing program. If the size of the remapping table could be kept small (on the order of 100 bytes), it would certainly be advantageous to update this table at different program phases. The table reload cost would be offset by the execution time savings due to reduce I-cache misses.

Although the remapping tables in our study are not large, the sizes of these tables for 128 pattern implementations are probably too large to permit incorporating them as a component of program state (due to the high cost of loading them upon thread dispatch). We believe either of the above two implementation schemes could reduce the remapping tables to manageable proportions.

Throughout our studies, we have presumed a single instruction stream model for program execution. This could certainly be integrated with a VLIW program execution model. In this case, each line of the remap table could hold the line of instructions for parallel dispatch. Couple with trace scheduling [3], an entire high-incidence trace could be represented as a single opcode.

6. Bibliography

[1] A. Aho, R. Sethi and J. Ullman, *Compiler: Principles, Techniques and Tools,* Addison-Wesley, 1986.

[2] P. Bird and T. Mudge, *An Instruction Stream Compression Technique*, CSE-TR-319-96, EECS Department, University of Michigan, November 1996.

[3] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*, ACM Distinguished Dissertation, MIT Press, 1985.

[4] A. Eustace and A. Srivastava, *ATOM: A flexible interface for building high performance program analysis tools*, Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems, January 1995.

[5] "Intel's P6 Uses Decoupled Superscalar Design," Microprocessor Report **9(2)**, 16 February 1995.

[6] S. Perl and R Sites, St*udies of Windows NT performance using dynamic execution traces*, Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation, October 1996.

[7] E. Rotenberg, S. Bennett, and J. Smith, T*race Cache: a Low Latency Approach to High Bandwidth Instruction Fetching*, Proceedings of the 29th Annual International Symposium on Microarchitecture, December 1996

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

[9] R. Uhlig, *Trap-Driven Memory Simulation*, Ph.D dissertation, EECS Department, University of Michigan, Ann Arbor, MI, 1995

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

Appendix

This appendix contains the comparison of compressed instruction streams to uncompressed streams for the integer benchmarks of the SPEC benchmarks.

Note that the miss rate scaling factors for these sets of graphs change for each graph.