Evaluation of Extended Dictionary-Based Static Code Compression Schemes

Martin Thuresson and Per Stenstrom
Department of Computer Science and Engineering
Chalmers University of Technology
{*martin,pers*}*@ce.chalmers.se*

ABSTRACT

This paper evaluates how much extended dictionary-based code compression techniques can reduce the static code size. In their simplest form, such methods statically identify identical instruction sequences in the code and replace them by a codeword if they yield a smaller code size based on a heuristic. At run-time, the codeword is replaced by a dictionary entry storing the corresponding instruction sequence.

Two previously proposed schemes are evaluated. The first scheme, as used in DISE, provides operand parameters to catch a larger number of identical instruction sequences. The second scheme replaces different instruction sequences with the same dictionary entry if they can be derived from it using a bit mask that can cancel out individual instructions. Additionally, this paper offers a third scheme, namely, to combine the two previously proposed schemes along with an off-line algorithm to compress the program. Our data shows that all schemes in isolation improve the compressibility. However, the most important finding is that the number of operand parameters has a significant effect on the compressibility. In addition, our proposed combined scheme can reduce the size of the dictionary and the number of codewords significantly which can enable efficient implementations of extended dictionary-based code compression techniques.

Categories and Subject Descriptors

C.1 [**Computer System Organization**]: Processor Architectures; B.3 [**Hardware**]: Memory Structures

General Terms

Algorithms, Design, Experimentation, Measurement, Performance

Keywords

Code Compression, Code Size Reduction, Dynamic Decompression, Memory Size Reduction

1. INTRODUCTION

Designers of many embedded systems, especially mobile appliances, often face challenging design tradeoffs owing to form factor requirements, limited battery capacity, and small cost margins. The trend towards more functionality in these appliances, makes the memory size an especially important factor in reducing energy consumption, chip area, and cost. We focus in this paper on techniques to reduce the static code size that combine dynamic and static approaches.

Static code size can be reduced by compressing individual instructions or sequences of instructions or both. Beszédes *et al.* [1] survey static code size reduction techniques. To position the contributions of this paper, let's first put the different approaches in perspective starting with techniques that compress individual instructions.

Many processors for the embedded domain extend the architecture with an option to switch to a reduced instruction set mode using a smaller instruction word size. MIPS-16 [8] and Thumb [15] both use 16 bits for each instruction. While such an approach can greatly reduce the static code size, one disadvantage is that the limited instruction size gives constraints on the number of registers each instruction can access as well as the size of the immediate values. Recently, attempts have been made to exploit variable-size instruction words to code more frequently occurring instructions with fewer bits. Lefurgy *et al.* [12] have looked at the possibility to compress common instructions with small codewords. CodePack [14, 7] divides each instruction into two parts and compresses each part separately using different static Huffman tables. However, variable-sized instruction words require extra care when branching since instructions are not always aligned in memory.

Continuing with compiler approaches, for a given ISA, the compiler can eliminate redundant code using a variety of code compaction methods [4] such as dead code elimination, code factoring, as well as dedicated transformations such as code abstraction. The advantage of static techniques is that they impose no performance overhead at run-time.

We focus in this paper on a complementary approach in which *identical* instruction sequences in the code are identified statically, after code generation, and replaced by a codeword if it yields a smaller code size based on a heuristic. At run-time, the codeword is replaced by a dictionary entry storing the corresponding instruction sequence.

DISE [3] generalizes the notion of an identical instruction sequence by extending the codeword with operand parameters. As a result, code sequences that only differ in the set of operands used can use the same codeword but with different parameters. A critical design choice is of course how many operands should be used. This is one of several design choices studied in detail in this paper. A second generalization of the basic technique is to replace different code sequences with a generic code sequence and a bitmask. The idea is that a replaced code sequence is a subset of the generic code sequence and the bitmask designates the subset. This approach is called *Bitmask Echo* and was proposed by Lau *et al.* [9].

This paper systematically evaluates the design space of these extended dictionary-based code compression techniques by focusing on the compressibility achieved by each technique in isolation. Especially, we look in detail to what extent the number of parameters and the size of the bitmask impact on the compressibility, the dictionary size, and the number of codewords. Previous work on DISE shows that adding operand parameters enables efficient compression though no detailed study has to our knowledge been published nor has the implication of using a bitmask been analyzed in this framework.

Apart from our findings that the number of operand parameters has a significant impact on the compressibility, we also contribute with a generalized architectural framework and an off-line algorithm for identifying identical code sequences using a combined technique that uses both operand parameters – according to DISE – and bitmask – according to Bitmask echo. This combined technique is shown to significantly reduce the size of the dictionary as well as the number of codewords which may yield more efficient implementations of extended dictionary-based compression techniques.

The paper is organized as follows: In Section 2, we introduce the architectural framework and describe in detail the different techniques evaluated later. Section 3 then describes the combined technique and its algorithm for compressing the static code size. We then introduce the experimental methodology in Section 4 and present the experimental results in Section 5. Finally Section 6 discusses related work and Section 7 presents the conclusions.

2. ARCHITECTURAL FRAMEWORK AND PRIOR ART

Dictionary-based compression techniques are based on the simple idea that recurring (static) sequences of instructions need not to be stored more than once. The repeated sequences of instructions are placed in a dictionary and assigned an ID. Compression is achieved by substituting the sequence of instructions in the program with a pointer to the dictionary entry, denoted a (*codeword*). As an example, consider the instruction sequence `ABCDABC`. Here `ABC` could be stored in a dictionary, assigned the ID 1 and the program would then simply be `(1)D(1)`, where `(1)` is the codeword associated with ID 1.

An architecture for decompressing dictionary-based instruction streams is shown in Figure 1. At an instruction decode, when a codeword is identified by the codeword matching-logic, the instruction is read from the dictionary instead of the normal instruction stream. This framework was introduced in DISE [3] and we will host many variations of the basic scheme in this framework.

We now extend this baseline method with two previously proposed generalizations: First, *parameterized schemes* aim

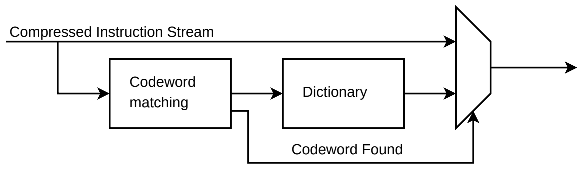


Figure 1: Architectural framework for extended dictionary-based static code size compression techniques.

at using the same codeword not only for sections of code that are identical, but also for sections of code that are identical disregarding a set of *operand parameters*. Second, *bitmask schemes* allow different sequences that are subsets of the same *generic* sequence to share that sequence by using a bitmask that designates which subset of the generic sequence that makes up each sequence. We describe each of these schemes in detail in the next section.

In Figures 2(a) and 2(b), we show the potential of combining both of these schemes by using two code sequences found in the PGP benchmark available in Mediabench and shown in Figure 2(a). The dictionary entry in Figure 2(b) can represent all instructions in Figure 2(a) in the following way. The resulting codewords are shown in the bottom of Figure 2(b).

The first codeword is expanded with the operand parameter P1 being .LL19 and the bitmask cancels out the *mov* instruction in the dictionary entry. The second codeword assigns the parameter .LL155 and the bit mask cancels out the first and the last instruction. Let's now study each of the techniques in isolation.

2.1 DISE - Dynamic Instruction Stream Editing

DISE [3] is a static and dynamic scheme that allows a programmer to implement general application customization functions. The hardware consists of a pattern-matching unit in the instruction-fetch pipeline that expands codewords to uncompressed instructions, thus allowing the user to dynamically edit the instruction stream. While this general approach has many applications, we focus here solely on its use to compress the static code size.

Figure 3 shows the DISE architecture and how it is used for static code compression where codewords are replaced with instruction sequences from the dictionary. The codeword logic simply detects if an instruction is a codeword which then forces the dictionary to generate the appropriate instruction sequence.

A novel idea in DISE is that it allows operand parameters in the codewords. In this way, DISE manages to use the same codeword and dictionary entry to similar, but not identical, sections of instructions. In particular, two instruction sequences that only differ in *e.g.* the choice of source operands, can use the same codeword with a set of parameters designating the source operands.

Clearly, the size of the dictionary is critical as it sits in the front end of the pipeline. Its size is affected by the number of codewords as well as the length of the instruction sequences associated with a codeword. The approach taken in DISE is to use a dictionary cache, (called *Pattern Table* and *Replacement Table* in the DISE architecture) that stores

```
Uncompressed Program
  add    %o0    -1   %o0
  cmp    %o0     0
  bge   .LL19
  st     %o0  [%i1]
  ..
  cmp    %o0     0
  bge   .LL155
  mov    %o0   %l0
```

(a) Original Code

```
Dictionary
DE1  add    %o0     -1   %o0
     cmp    %o0      0
     bge     P1
     mov    %o0    %l0
     st     %o0   [%i1]

Compressed Program
(DE1  .LL19  11101)
..
(DE1  .LL55  01110)
```

(b) Compressed Program.

Figure 2: Example of code compression from the PGP benchmark (Mediabench) using extended dictionary-based compression techniques.

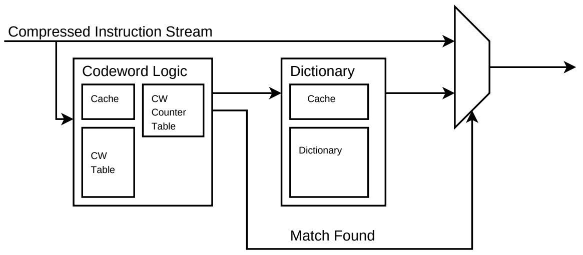


Figure 3: The DISE architecture: Expansion of codewords using dictionary caches.

a subset of a larger table. On the other hand, the codeword logic (see Figure 3) is not trivial to extend with a cache; if a match is found in the codeword cache, we can raise the *Match Found* signal, but if no match is found, a match *could* be in the larger table. An extra table, the *CW Counter Table*, is used to keep track of how many codewords that could possibly match the OP-field of the instruction. If all these are already in the cache, a miss can be flagged, otherwise all possible codeword patterns are loaded into the cache, and the instruction is checked again.

Extending the dictionary with a cache is trivial since a miss in the dictionary cache means that the dictionary entry must be in the dictionary. However, cache misses come with a cost in performance. Many codewords and large dictionaries may cause a higher miss rate and thus performance overhead; thus there is a tradeoff between performance and compression ratio [3]. Bottom line is that the number of codewords needed and the size of the instruction sequences are critical to efficient implementations of this scheme. We will evaluate the impact of these parameters later in the paper.

2.2 Echo and Bitmask Echo

Fraser introduced an instruction which allows the direct interpretation and execution of programs compressed in a LZ77[1]-like fashion [5]. This instruction, called *echo*, repeats a sequence of instructions at a given offset and length from the current execution point. A difference from traditional dictionary-based compression schemes is that no separate dictionary is used since echo instructions identify sequences inside the program. The work assumes interpretation of byte-code although it also discusses other types of implementations.

As a follow-on study, Lau *et al.* [9] considered a hardware implementation of the echo instruction, and also extended it by allowing bitmasks instead of length to allow the merger of similar, though not identical sections of code. The bitmask is used to generate all instruction sequences that are a subset of the coded instruction sequence.

By using *Bitmask Echo*, they achieve on average a compression ratio (size of compressed code over uncompressed code) of about 0.85 for applications from the Mediabench suit compiled for the Alpha ISA. The compression ratios of echo and bitmask echo can not be directly compared in these studies mentioned above since the original echo instruction used byte code, which is believed to be easier to compress than register-based instructions. We will evaluate bitmask echo in the same framework as DISE.

3. COMBINED EXTENDED DICTIONARY-BASED COMPRESSION SCHEME

In both Bitmask Echo and DISE, additional compression is achieved by extending the baseline scheme with parameters allowing sequences of similar instructions to be represented using one codeword.

We propose an extension to DISE which designates one of its parameters as a bitmask instead of an operand parameter, creating a flexible framework with highly parameterizable dictionary entries. By adding information in the dictionary about how the operands should be interpreted, all three operands can be used as operand parameters when pre-

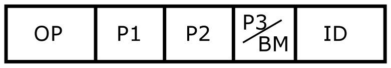


Figure 4: Proposed codeword format for the extended dictionary-based compression scheme.

[1]general sliding window data-compression algorithm

```
1   dictionary ← {} // Create empty dictionary
2   do
3       // Initialize the set of all possible dictionary entries
4       all_possible_des ← {}
5       // Iterate through all possible instruction sequences
6       for start ← 0 to program_size
7           for size ← min_de_size to max_de_size
8               // Add sequence only if one entry point exist and if
9               // no codewords is inside sequence
10              if valid_sequence(program, start, size) then
11                  all_possible_des ← all_possible_des ∪ {(start, size)}
12      if all_possible_des ≠ {} then
13          // Find sequence with best immediate compression,
14          // returns NULL if no sequence yielding compression exists
15          best_pde ← get_best_pde(all_possible_des, program)
16          if best_pde ≠ NULL then
17              // Update dictionary and replace instructions with codewords in program
18              dictionary ← dictionary ∪ {best_pde}
19              program ← update_program(program, best_pde)
20  // Continue until no more compressible sequences can be found
21  while all_possible_des ≠ {} and best_pde ≠ NULL
```

Figure 5: Pseudo code for generating dictionary entries.

ferred. In our proposed codeword format, shown in Figure 4, the OP-code triggers the replacement; P1, P2, and P3 are parameters that can be used as operands in the codewords, and ID identifies the dictionary entry in the replacement table. BM is the bitmask used to cancel out instructions in the dictionary entry. In this study it is assumed that the cost of the dictionary entry is the number of instructions it holds.

3.1 Baseline Scheme

Generating an optimal encoding using dictionary-based compression (even without parameters) is NP-complete in the size of the code [13]. The baseline algorithm used to generate the dictionary is based on the greedy algorithm described by Lefurgy *et al.* [12]. The pseudo-code in Figure 5 gives an overview of the algorithm. In each iteration, one dictionary entry is selected and the program is updated. When no more dictionary entries can be found that improves the compression ratio of the program, the algorithm terminates and outputs the compressed program. Each program is compressed independently and uses its own dictionary.

The first part of the algorithm (line 2 to 11) generates the set of all possible dictionary entries that will be considered when selecting the next sequence to be included in the dictionary. The second part (line 12 to 20, the function *get_best_pde* selects the one that according to our algorithm should yield the best compression, and the selected sequence is added to the dictionary and the program is updated.

The heuristic used to choose the best possible dictionary entry, *immediate compression*, is defined in Equation 1 and weights the cost of the dictionary entry and the codeword against the gain from the removed instructions. In the equation, *Entry Size* means the number of instructions in the dictionary, *Number of Codewords* shows how many times the codeword is used in the program, and *Removed Instructions* gives the number of instructions that were replaced with this codeword. The implications of using another function when selecting the best dictionary entry is analyzed in Section 5.4.

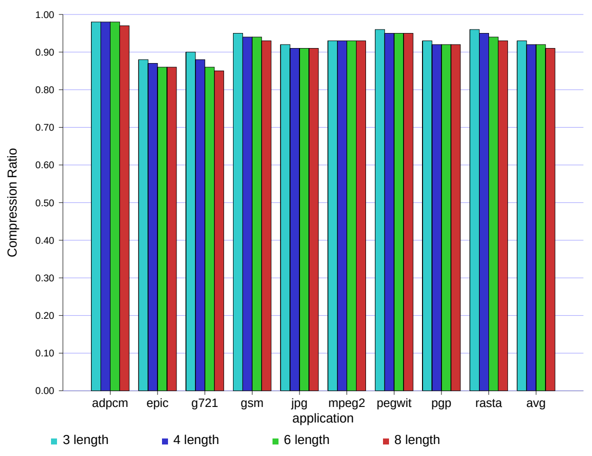
$$Immediate\ Comp. = \frac{Entry\ Size + No.\ of\ Codewords}{Removed\ Instructions} \quad (1)$$

To avoid implementation complexity, jump and branch instructions are only allowed to jump to the first instruction of a dictionary entry. This constraint limits the number of instruction sequences to consider in the algorithm. The maximum size of the dictionary entry, *max_de_size*, is also an input to the algorithm. While previous algorithms only consider basic blocks [9, 3], our implementation allows superblocks, a more general form of instruction sequences that can have multiple exit-points.

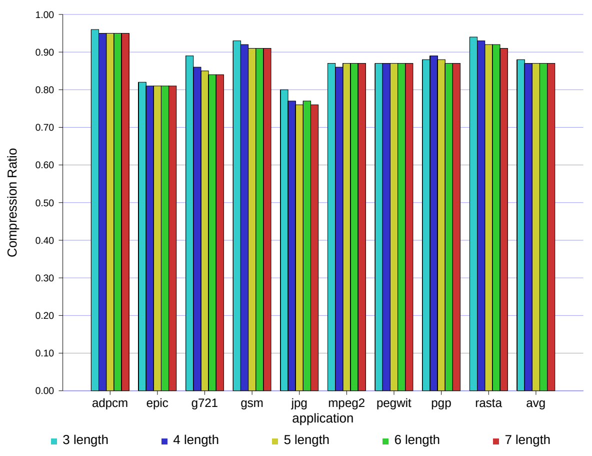
3.2 Algorithm for Parameterized Dictionary Entries

When calculating the compression achieved using a particular instruction sequence (possible dictionary entry), the baseline algorithm has to be extended to allow operand parameters. In order to match as many instructions as possible, operand parameters are added to the possible dictionary entry when calculating the possible compression ratio (line 15 in the algorithm in Figure 5). Again, a greedy algorithm is used; when comparing if an instruction sequence could be replaced by a codeword, parameters are always added to the possible dictionary entry if it helps locally.

To make use of bitmask parameters, we have further extended the baseline algorithm. When analyzing the gain from using a given possible dictionary entry (with at most two operand parameters), we can still use the codeword even though not all instructions in the entry exist since a bitmask can cancel them out. If the possible dictionary entries are generated the same way as before, we might lose some possibility of compression. For example, no codeword exists that can represent this sequence of instructions: `ABDEACDE`. Given that the maximum size of the dictionary is 4, the following *possible dictionary entries* are then generated by the baseline algorithm: {`AB, ABD, ABDE, BD, BDE, BDEA, DE,`



(a) Maximum length of dictionary entry is varied



(b) Maximum size of bitmask is varied

Figure 6: Compression ratio when (a) no parameters are used, and (b) when only bitmask is used.

`DEA, DEAC, EA, EAC, EACD, AC, ACD, ACDE, CD, CDE`}. None of these can be used as a codeword to achieve any compression.

We extend the baseline algorithm by adding sequences of instructions not found in the program that may be used on several places with the help of bitmasks. Already present possible dictionary entries are used as templates and are extended with extra instructions. With this extension, the following possible dictionary entries can also be considered: {`ABCDE, ACBDE`}. Using one of them makes it possible to compress the program. The dictionary could contain the entry `ABCDE` with ID 1, and the program would be (1_{11011}) (1_{10111}), where bitmasks are denoted with subscripts.

The insertion of new possible dictionary entries can be done in two different ways. The algorithm described in Figure 5 is modified at either line 11 or line 17. In the first case, new possible dictionary entries may be generated based on each valid sequence in the program, while in the latter case the new entries are only based on one entry. When applicable, in our experiments we use both and select the best result from the two. Since the modification at line 11 added more possible dictionary entries, it usually gives the best result at the cost of higher computational complexity.

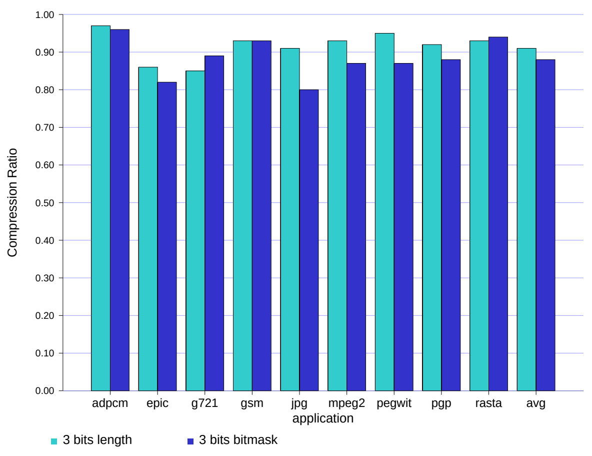
4. EXPERIMENTAL METHODOLOGY

To evaluate how the different parameters affect the compression ratio, the number of codewords, and the size of the dictionary, applications from the Mediabench [10] suite were compressed using our proposed algorithm. Table 1 describes the benchmarks used in the experiments. The programs were compiled using *gcc* to Sparc ISA. As Mediabench is a collection of real applications, each application was compiled with the settings recommended by the original authors. All of them used the optimization flag and most of them used -O2. The output from the algorithm is a compressed program with statistics about the size of the program, dictionary and the number of codewords used.

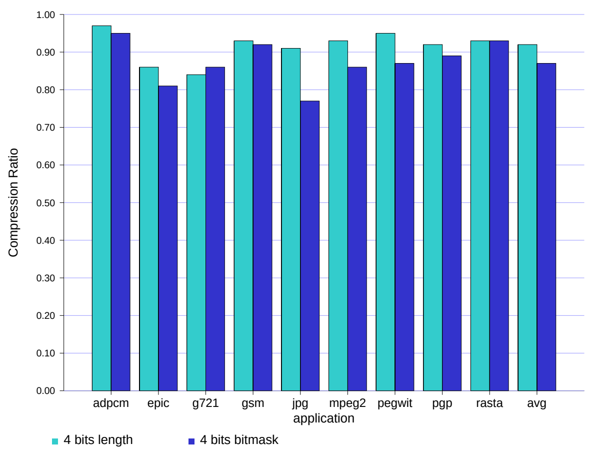
The rather high computational complexity of the algorithm mentioned in the previous section limits the input size that can be handled in a reasonable time. To get a fair comparison we chose to consider one thousand instructions from each of the applications as input in all our experiments, except for ADPCM which only contains 365 instructions. We expect that the compression ratios achieved are conservative for two reasons: First, experimentation with different code sequences in the same application containing thousand instructions yielded consistent results. Second, larger code sequences would yield higher compression ratios making our results conservative.

Name	Description	Number of Instructions
ADPCM	Simple audio encoding/decoding	365
EPIC	Wavelet-based image compression utility	5821
G.721	Voice Compression Algorithm	1648
GSM	Full Speech Transcoding	8696
JPEG	Lossy image compression algorithm	32926
Mesa	3-D graphics library clone for OpenGL	113611
MPEG	Algorithms for high quality digital video processing	25433
PEGWIT	Public key encryption and decryption	10794
PGP	Public key encryption and decryption	37791
RASTA	Program for speech recognition	10770

Table 1: Selected programs from the Mediabench used in the experiments.



(a) Maximum 3 bits for either length or bitmask.



(b) Maximum 4 bits for either length or bitmask.

Figure 7: Compression ratio when comparing no parameter with bitmask parameter. When no parameter is used, the maximum size of dictionary is fixed.

5. EXPERIMENTAL RESULTS

First we consider the impact of the different parameters separately before considering their combined impact. In each diagram, the bar labeled *avg* shows the average across all the benchmarks.

5.1 Impact of Bitmask Parameters

The first experiment considers the gain from adding a bitmask to a parameterless version of our general architecture. In bitmask echo [9], the bitmask field replaces a length field and here we study the gain from doing this in our architecture. Figure 6(a) shows the compression ratio achieved when changing the maximum length of the dictionary-entries (*max_de_length* in Figure 5) from three to eight. The average compression ratio, presented in the rightmost group of the diagram, shows a total improvement in compression ratio of about two percent.

When using a bitmask parameter in a codeword, the maximum size of the bitmask also gives the maximum size of the dictionary entry. The next experiment adds a bitmask parameter to the codeword and considers how different maximum sizes of the dictionary entry change the compression ratio. Figure 6(b) shows the compression ratio when the bitmask size is changed from three to seven bits. Note that this scheme is similar to the bitmask echo, though in our framework, the instructions are stored in a dedicated dictionary. In almost all cases the compression ratio is improved when moving to a larger bitmask. The few cases where the compression ratio is worsened can be attributed to our greedy algorithm, that makes locally good choices which turns out to be worse in the long run. The small improvement in compression ratio when allowing larger dictionary entries suggests that it is better to use small dictionary entries if it results in a smaller dictionary in total, which it in most cases does in the experiments here.

Comparing the results in Figure 6(a) and 6(b) we can see that adding bitmask-enabled codewords gives an improved compression ratio. A fair comparison between a parameterless technique and one with bitmask is to compare the same maximum possible size of dictionary-entry for both. Figure 7 compares the case assuming a fixed dedicated size for length or bitmask. Using three bits for bitmask or length achieves better or equal compression ratio in all but two benchmarks, Figure 7(a). When using up to four bits, the result is even better for bitmask, on average, 5% better as seen in Figure 7(a). ADPCM has the worst compression ratio of the benchmarks. This can be explained by the fact that it only contains 365 instructions, making it harder to find useful dictionary entries.

The results in this section show that bitmask parameters enable improved compression ratio to the general dictionary based compression scheme. Another important finding is that the bitmask length does not affect the compression ratio in a major way.

5.2 Impact of Operand Parameters

DISE extends the base dictionary method with operand parameters. The default implementation allows three operands, each of them which may be register or immediate values. This experiment considers how the number of parameters affects the compression ratio when only operand parameters are allowed, i.e., with no bitmask. Figure 8 shows the compression ratio when the number of parameters is varied from zero to three.

The result shows that the number of parameters has a quite large impact on the compression ratio. The last bar in the diagram shows that moving from zero to three parameters improves the compression ratio by 18% on average. With each extra parameter, the gain gets smaller and smaller. An interesting thing to note is that even though the maximum number of parameters is fixed, the actual number of parameters a codeword uses varies between zero and the maximum. For the compressed programs which allow three parameters, only about half use all of them. For the bench-

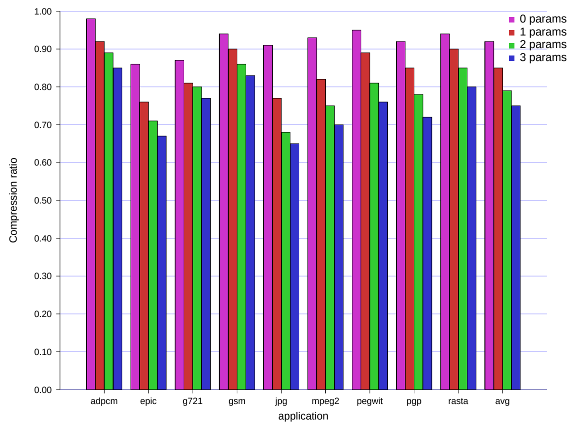


Figure 8: Variation of number of operand parameters

marks considered, the number varies between 45% and 75%, with an average of 55%.

During this study we assume that one operand parameter may be referenced several times in the codeword. The next experiment compares this to the case when a parameter is only allowed to be used once in the dictionary. The result can be seen in Figure 9. As expected, allowing the same parameter several times has a positive effect on the compression ratio. This is intuitive since neighboring instructions often use the same set of registers.

The assembly code in Figure 10(a) is taken from the benchmark EPIC. Even though the two sections of code differ in six operands, only three parameters are needed to make them identical when we allow parameter reuse. Figure 10(b) shows a dictionary entry that can replace both sections of code and the codewords in the compressed program. All experiments with operand parameters use parameter reuse unless otherwise noted.

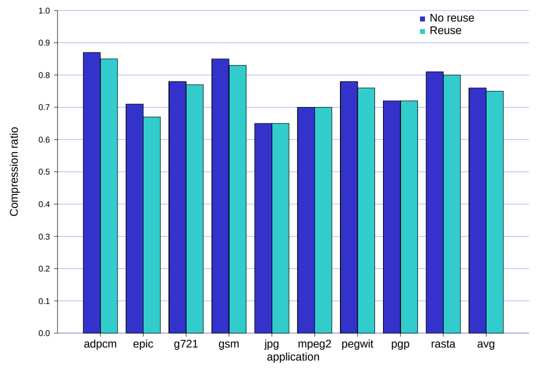


Figure 9: Impact of operand-parameter reuse on the compression ratio.

A conclusion that can be drawn from this section is that each extra operand parameter that is allowed gives us improved compression ratio. Also we note that a big fraction of dictionary entries use less than the maximum allowed parameters.

```
ld    [%o7+4]  %o0
sra       %o4    1   %o4
add       %o0  %o4   %o0
...
ld    [%l1+4]  %o2
sra       %o4    1   %o3
add       %o2  %o3   %o2
```

(a) Original Code

```
Dictionary
DE1  ld   [P1+4]   P2
     sra     %o4    1    P3
     add      P2   P3    P2

Program
CW   %o7     %o0   %o4   DE1
..
CW   %l1     %o2   %o3   DE1
```

(b) Compressed Program.

Figure 10: Example code taken from EPIC which could only be compressed if parameter reuse is allowed.

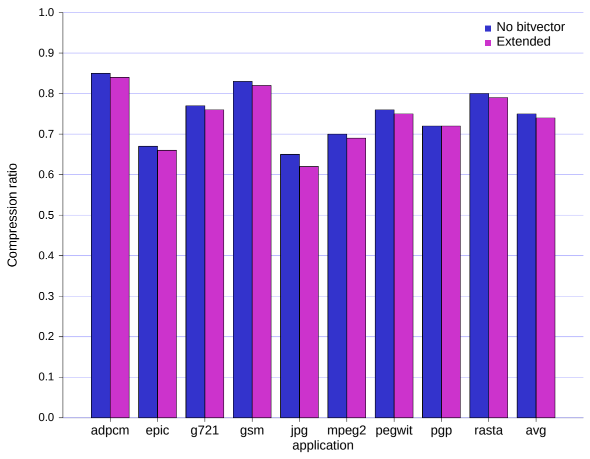
5.3 Impact of Operand and Bitmask Parameters

By allowing one of the parameters to be either an operand-parameter or a bitmask it is possible to extend the DISE scheme to use bitmask when it is useful and to use three operand parameters when that gives more compression. Since the encoding in the dictionary shows how the parameters should be interpreted, it is possible to get at least the same compression as the case without a bitmask. The fact that many of the dictionary entries do not use all the parameter, as described in Section 5.2, suggests that we could achieve better results using this approach.

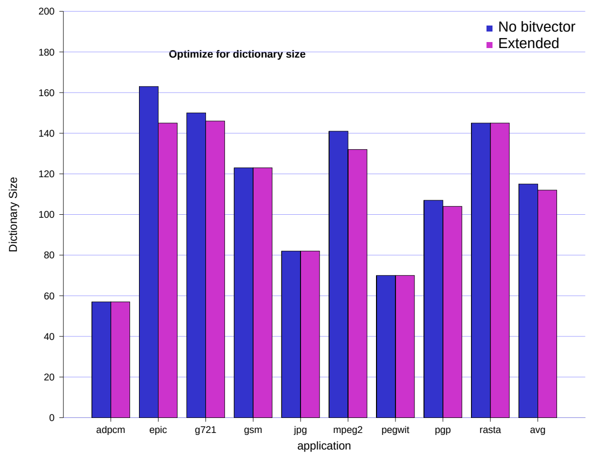
When not using bitmask, it is possible to generate the dictionary entries by only looking at consecutive instructions from the program. When the bitmask can be used, we have analyzed different methods of generating dictionary entries by looking at a larger domain of possible dictionary entries. The method that gave the best result was to start by looking at sequences that exist in the program, and then adding instructions to these if it yielded better results, as outlined in Section 3.2.

The results in Figure 11(a) shows only small improvements in the compression ratio. On the other hand, when looking at the size of the dictionary, Figure 11(b), and the number of codewords, Figure 11(c), used in the program we see that they have decreased substantially for some of the applications. This has the positive effect of decreasing the number of misses in both the codeword- and dictionary-cache in our extended DISE-architecture. Another positive side-effect is that if the compiler is compression-aware it has one more tool to efficiently schedule instructions to be compressed.

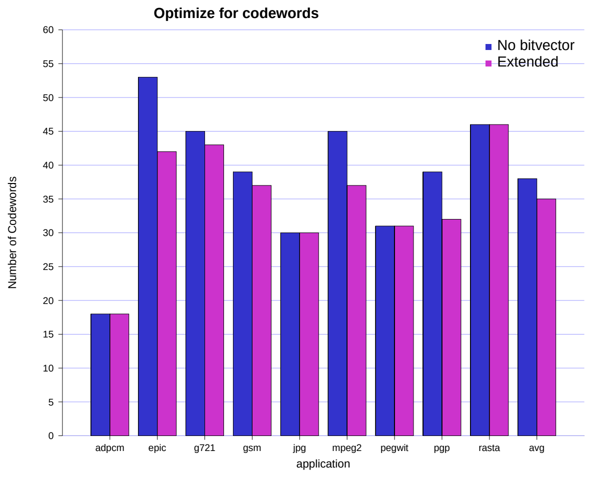
When analyzing the effect of parameter reuse in this ex-



(a) Total Size



(b) Dictionary Size



(c) Number of Codewords

Figure 11: The diagrams show the gain when moving from only operand parameters to one with both operand and bitmask parameters.

periment, the result was similar to the experiment with only operand parameters in Section 5.2, though we noted that more applications gained from it in this configuration. In our previous experiment, Figure 9, the compression ratio could only be improved when the number of parameters for a possible dictionary entry could be reduced to three or less, but when allowing a bitmask, reducing it to two or less also makes it possible to use one operand as a bitmask parameter.

Figure 12 shows the compression ratio for our architecture when the number of allowed operand parameters is varied. A difference between this result and the case when no bitmask was used (Figure 8) is that the compression ratio is increased most when changing from two to three operand parameters.

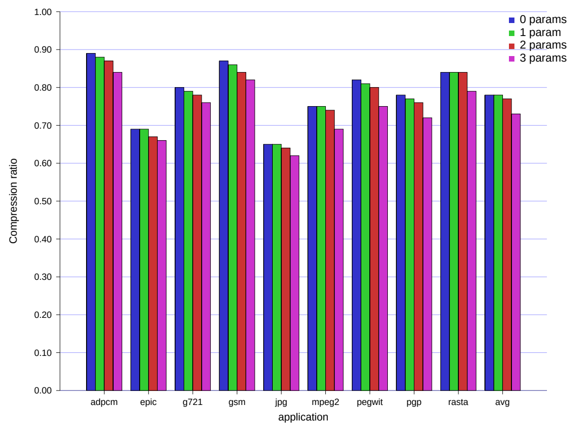


Figure 12: Compression ratio for bitmask parameter and different number of operand parameters.

5.4 Immediate and Absolute Compression

As described in Section 3.1, immediate compression has been the metric used to select the best possible dictionary entry in the function *get_best_pde*, Figure 5. Another cost function analyzed was *absolute compression*, Equation 2. Here *Size after comp.* refers to the compression ratio achieved if this possible dictionary entry is chosen.

$$Abs.\ Comp. = \frac{Size\ before\ comp.. - Size\ after\ comp.}{Size\ before\ compression} \quad (2)$$

Figure 13 compares the compression ratio achieved using the two cost functions. Immediate compression always results in better or equally good compression ratio as absolute compression. The absolute compression metric sometimes selects too greedily, selecting small dictionary entries that are used in many places, making it harder to find dictionary entries in the next iterations.

6. RELATED WORK

Compiler techniques such as Code Factoring target the same type of redundancy as dictionary based compression. Code Compaction is the general term for code compression techniques that require no decompression to execute. Code Factoring tries to use only one representation for instructions that are similar and find ways to execute them without

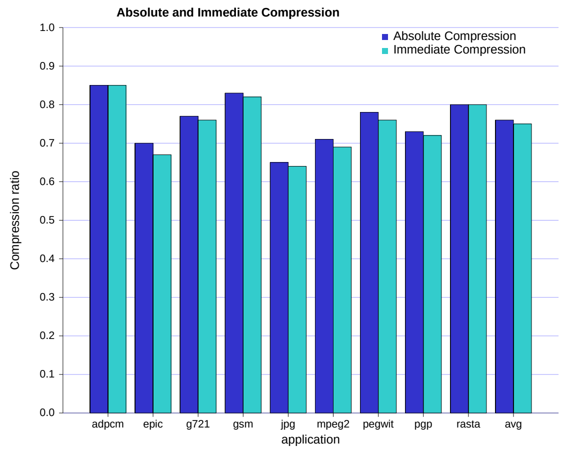


Figure 13: Absolute and Immediate-compression as a cost-function

much overhead. This can be done using jump instructions (procedural abstraction) or moving instructions to common branches (local code transformations). Debray *et al* [4] studied a way of abstracting partially matched blocks, but because of low benefit and high computational complexity, it was not considered a beneficial transformation in most cases. Code Compaction may be used at the same time as hardware dictionary-based compression since code compaction is a pure software method.

CodePack [14, 7] is a compression scheme used by IBM in the PowerPC 405. CodePack works between the L1-cache and the rest of the memory hierarchy and is designed to make the CPU-core unaware of the compression. It works by compressing fixed length instructions to variable length codewords. Each 32-bit instruction is divided into two 16-bit parts which is encoded using two separate pre-computed Huffman-tables. Using CodePack, IBM reports a compression ratio of 0.6 with a performance change within 10% compared to execution of an uncompressed program [6]. CodePack extends the work done on variable-length codewords in dictionary-based instruction compression by Lefurgy [11]. An important difference compared to dictionary-based schemes is that CodePack compresses individual instructions while our scheme compresses sequences of instructions. Because of the different granularities targeted, CodePack can be used at the same time as our proposed dictionary based compression scheme. Combining CodePack with Bitmask Echo results in an additional 7% improvement in compression ratio [9].

Collin and Brorsson[2] also uses variable length instructions in order to compress the static code size. Their instructions are byte aligned and their main focus is to reduce the power consumption in the instruction-fetch pipeline-stage. As in our proposed architecture, the beginning of the pipeline needs to be modified. Using their approach, they achieve up to 15% energy savings in the data path and memory system while compressing the program up to 30%.

7. CONCLUSION

This paper has studied several schemes of extended dictionary-based code compression. Targeting the embedded domain where static code compression can be an important tool, programs from the Mediabench suite has been used to evaluate such schemes. Compared to a baseline dictionary based-code compression scheme, bitmask-enabled codewords results in 5% better compression ratio while operand-enabled codewords achieve about 20% improvement. Our proposed framework and algorithm for combing both schemes achieves at least the same reduction in code size as operand parameters, but enables more efficient coding of the dictionary allowing for a more efficient implementation. Our study on heuristics and cost-functions in the algorithms also shows that immediate compression performs better than absolute compression and that register reuse enables more efficient use of the parameters.

Acknowledgments

The authors are indebted to Magnus Själander and the other members of the FlexSoC project for comments. We also thank the anonymous reviewers. This research has been sponsored by the Swedish Foundation for Strategic Research (SSF) under the FlexSoC program.

8. REFERENCES

[1] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.

[2] M. Collin and M. Brorsson. Low power instruction fetch using profiled variable length instructions. In *SOC Conference, 2003. Proceedings. IEEE International*, pages 183–188, 2003.

[3] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. DISE: a programmable macro engine for customizing applications. In Doug DeGroot, editor, *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, volume 31, 2 of *Computer Architecture News*, pages 362–373. ACM Press, June 2003.

[4] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, 2000.

[5] Christopher W. Fraser. An instruction for direct interpretation of LZ77-compressed programs. Technical Report MSR-TR-2002-90, Microsoft Research, Microsoft Corperation One Microsoft Way Redmond, WA 98052, USA, September 2002.

[6] Mark Game and Alan Booker. CodePack: Code compression for PowerPC processors. version 1.0. Technichal whitepaper, IBM Microelectronics Division, May 2000.

[7] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0.* International Business Machines (IBM) Corporation, 1998.

[8] Kevin D. Kissell. MIPS16: High-density MIPS for the embedded market. Technical report, Silicon Graphics MIPS Group, 1997.

[9] Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, and Brad Calder. Reducing code size with echo instructions. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, pages 84–94. ACM Press, 2003.

[10] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[11] Charles Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, 2000.

[12] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. Technical Report "CSE-TR-342-97", EECS Department, University of Michigan, 8 1997.

[13] J. Storer. NP-completeness results concerning data compression. Technical Report 234, Department of Electrical Engineering and Computer Science, Princeton University, 1977.

[14] D. J. Auerbach J. D. Harper T. M. Kemp, R. K. Montoye and J. D. Palmer. A decompression core for PowerPC. *IBM Journal of Research and Development*, November 1998.

[15] James L. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4), March 1995.