Adaptive Object Code Compression

John Gilbert
Compiler Design Research Group
School of Computer Science and Statistics
Trinity College Dublin
gilberj@cs.tcd.ie

David M Abrahamson
Compiler Design Research Group
School of Computer Science and Statistics
Trinity College Dublin
david.abrahamson@cs.tcd.ie

ABSTRACT

Previous object code compression schemes have employed static and semiadaptive compression algorithms to reduce the size of instruction memory in embedded systems. The suggestion by a number of researchers that adaptive compression techniques are unlikely to yield satisfactory results for code compression has resulted in virtually no investigation of their application to that domain. This paper presents a new adaptive approach to code compression which operates at the granularity of a program's cache lines, where the context for compression is determined by an analysis of control flow in the code being compressed. We introduce a novel data structure, the compulsory miss tree, that is used to identify a partial order in which compulsory misses will have occurred in an instruction cache whenever a cache miss occurs. This tree is used as a basis for dynamically building and maintaining an LZW dictionary for compression/decompression of individual instruction cache lines. We applied our technique to eight benchmarks taken from the MiBench and MediaBench suites, which were compiled with size optimization and subsequently compacted using a link-time optimizer prior to compression. Results from our experiments demonstrate object code size elimination averaging between 7.7% and 18.3% of the original linked code size, depending on the cache line length under inspection.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: Data compaction and compression—*program representation*; C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems

General Terms

Algorithms, Experimentation

Keywords

Code compaction, code compression, code size reduction

1. INTRODUCTION

Object code compression is a technique used to reduce the cost of embedded systems. Its benefits include a reduction in the quantity of physical memory required for system implementation, and an associated reduction in system level power consumption due to the decreased bandwidth requirements for accessing external memory where compressed code is stored.

Compression is not generally applied to the code in a program as a whole. If it were, we would need to decompress the entire program at load-time resulting in no memory size saving. For this reason, and given that control passes from one part of a program's code to another while it executes, previous work has allowed decompression from arbitrary branch targets or instruction cache line boundaries in the code at run-time [2].

The majority of the schemes developed to date have compressed either individual instructions, fixed-size cache lines, or short variable-length instruction sequences which do not extend beyond the boundary of a basic block. Most of the schemes developed at this granularity have employed static or semiadaptive compression algorithms. Adaptive algorithms have been rejected for use due to the pervasive belief that they are mismatched for use with the small units of code which must be decompressed at run-time [12, 15, 16, 17, 19, 25, 26].

In this work we exploit the observation that decompression from *arbitrary* branch targets is not required, it is sufficient to be able to start decompression from each branch target *as it is encountered during execution of the program.* Using this knowledge we identify additional context to adaptively build a model for use when compressing, and decompressing, instruction code. Our approach works at the granularity of fixed-length program cache lines, determining the context for coding by object code control flow analysis. Our work presents a replacement compression technique for use in an architecture of the type described by Wolfe [25], referred to as a *Compressed Code RISC Processor.*

The remainder of this paper is structured as follows: first, we review the classification of data compression algorithms and present the Huffman and LZW algorithms. Then we summarize the Compressed Code RISC Processor architecture suggested by Wolfe. This is followed by a description of employing LZW rather than Huffman coding in Wolfe's architecture. Next a review is made of some elementary topics from compiler design, specifically the immediate dominance relation between blocks in a control flow graph. With the prerequisite background reviewed, we present a new data structure termed the *compulsory miss tree* and describe its use in constructing an improved LZW-based compression technique. Finally, we relate our work to that already described in the literature, present our experimental results, identify future work in the area and draw our conclusions.

2. DATA COMPRESSION

Data compression techniques may be classified according to a number of facets which include their fidelity, model type and adaptivity. The fidelity of a scheme is either lossless or lossy. Lossless algorithms allow the original data be reconstructed perfectly after being compressed and subsequently decompressed. This contrasts with lossy algorithms which allow for some loss of precision or entire omission of fine detail in the input as part of the compression process in return for improved size reduction.

Models are described as either statistical or dictionary-based. In a statistical model the probability of each input symbol occurring is determined. This allows an allocation of variable-length binary codes to each symbol in the input, such that symbols with a high probability of occurrence are given short codes while less likely symbols are given longer ones. Dictionary-based models operate by moving commonly occurring sequences of symbols from the input to a dictionary (table). Each occurrence of one of these sequences may then be replaced by a shorter code word indexing the appropriate dictionary entry.

There are three adaptivity classifications: semiadaptive, static and adaptive. In a semiadaptive scheme the data to be compressed are first analysed in their entirety, an appropriate model is then built, and finally the data is encoded. The model is stored as part of the encoded data, as it is required by the decompressor to reverse the encoding. Static schemes are similar to this, but a representative selection of data is used to build a fixed model which is hard-coded into compressors and decompressors. This has the advantage that no model must be explicitly stored with the compressed data, but the disadvantage that poor compression will result if the model is not representative of data presented for compression. Finally, adaptive techniques commence coding with an empty (or statically determined) initial model, and update this model as the coding of each successive input symbol occurs. When decoding compressed data, the same initial model used for compression is constructed and is appropriately updated as each symbol is recovered. This not only removes the need for the model to be explicitly transmitted or stored with the encoded data, but also facilitates tailoring the model to the particular data being compressed.

When compressing object code it is imperative that the original data can be recovered without loss, a requirement also shared by text-compression applications where lossy algorithms have no place. Many object code compression schemes have borrowed methodology from the text compression community using a variety of statistical and dictionary-based techniques [3]. With regards to adaptivity however, focus has remained on static and semiadaptive techniques, and little attention has been paid to the class of adaptive algorithms. This is a direct result of allowing decompression commence at arbitrary branch targets or instruction cache line boundaries in the object code being compressed. To allow decompression begin at these positions requires that the adaptive model be reset before compressing/decompressing each block. Given the short length of these blocks the model could not adapt sufficiently to give rise to compression, leading to a widely held belief that adaptive techniques are inherently mismatched with the requirements of object code compression [12, 15, 16, 17, 19, 25, 26].

Symbol	Frequency	Huffman code
a	5	1
b	3	00
c	1	01

a,c,a,b,a,a,b,a,b $\mapsto$ 1,01,1,00,1,1,00,1,00

Figure 1: Example of Huffman coding: Huffman code; original data; and Huffman coded data

Index (LZW code)	Phrase
0	a
1	b
2	*uninit*
.	.
.	.
2^n	*uninit*

(Initial dictionary)

a,a,b,a,a,b,a $\mapsto$ 0,0,1,2,4
(Original data $\mapsto$ coded data)

Index (LZW code)	Phrase
0	a
1	b
2	a, a
3	a, b
4	b, a
5	a, a, b
6	*uninit*
.	.
.	.
2^n	*uninit*

(Dictionary after coding)

Figure 2: Example of LZW compression showing dictionary adaptation

2.1 Huffman coding

To construct a Huffman code the data to be compressed are analysed and a count of the frequency of occurrence of each symbol in the input alphabet $\sum$ (for example, each byte or word) is determined. Using this information, variable-length binary codes are assigned to input symbols so that those appearing frequently are given short codes while those appearing infrequently are allocated longer ones [3]. These variable-length codes are constructed so that the prefix-property holds, that is, no code is a prefix of any other code. When compressing data the original symbols are replaced by their corresponding Huffman codes and stored in the output. The output also contains a table containing the mapping from Huffman codes to the input alphabet symbols for use by the decoding algorithm. Hence Huffman coding is an example of a lossless, statistical, semiadaptive technique. Decoding occurs by sequentially matching Huffman codes to the encoded data and outputting the corresponding symbols. A simple example of Huffman coding is presented in Figure 1.

2.2 LZW

LZW [24] is a popular text-compression algorithm. It is classified as lossless, dictionary-based and adaptive. A piece of LZW-coded data is comprised of a sequence of fixed-length n-bit indices into a dictionary of 2^n phrases, which is constructed in the following way. Initially the dictionary contains only the individual symbols in the input alphabet $\sum$ (say the bytes 0..255). The dictionary is then searched for the longest phrase which is a prefix of the input data. The index for this phrase is output in the compressed encoding,

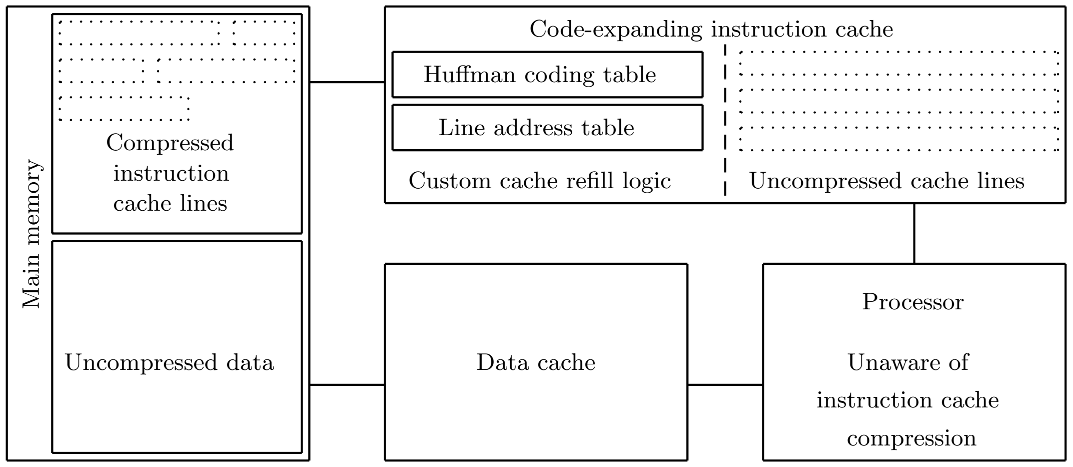
Main memory
Compressed instruction cache lines
Uncompressed data
Code-expanding instruction cache
Huffman coding table
Line address table
Custom cache refill logic
Uncompressed cache lines
Data cache
Processor
Unaware of instruction cache compression

Figure 3: Architecture of Wolfe's Compressed Code RISC Processor

and a new entry, consisting of the phrase just matched concatenated with the following input symbol, is inserted into the dictionary. Coding continues in this fashion, restarting at the next unmatched symbol in the input. When the entire dictionary is full no more entries can be added and the remaining input data is coded using the dynamically-constructed dictionary.

Decompression of LZW coded data starts with the same initial dictionary that was used for compression. Each codeword to be decompressed is used as an index to the dictionary and its corresponding phrase is output to the decompressed stream. Then the first symbol from the phrase in the dictionary indexed by the next codeword in the compressed stream is appended to the phrase just decompressed, and this sequence is then inserted as a new entry in the dictionary. In this way, the decompressor maintains the same dictionary as that generated during compression. The dynamically constructed LZW dictionary encodes a history of previously encountered phrases in the input stream and gives rise to compression when a single code (dictionary index) is output in place of multiple symbols from the input stream during coding. An example of LZW coding is presented in Figure 2, where the initial dictionary is shown on the top left, the data to be coded and the dictionary indices for its encoded form are shown on the bottom left and the final dictionary resulting from coding is shown on the right of the figure.

3. COMPRESSED CODE RISC PROCESSOR

Wolfe introduced the *Compressed Code RISC Processor* (CCRP) [25], an architecture where a standard RISC core is augmented with a code-expanding instruction cache (see Figure 3). Individual fixed-length cache lines are compressed to variable-length blocks using Huffman coding and stored in main memory. On a cache miss, the compressed cache line is loaded from memory, decompressed and placed in the cache by custom cache refill logic. As the location used by the processor for the requested cache line will differ from its compressed-form location in memory, due to the difference in size between the compressed and uncompressed forms of the program, the compressed cache line address in memory is looked up in a so called *line address table*. Once the cache line has been placed in the cache, the requested instruction is extracted and returned to the processor as if the data had never been compressed. Thus the decompression mechanism is entirely transparent to the processor.

The line address table (LAT) may be naively encoded using one entry for every cache line in the program. The entry contains the location, in main memory, where the compressed version of each cache line is located. In a system where cache lines are 32 bytes long and addresses are 32 bits long, 32 bits of storage for every 32 bytes of instruction data are required—that is, an increase of 12.5% over the original program size (assuming compressed cache lines are byte-aligned). A better encoding can be derived by allowing a single LAT entry represent the mapping for a number of cache lines using a simple delta coding scheme. Using a delta coding approach, a base address is stored for the first line and offsets are added to this to obtain the address for contiguous lines with higher addresses in compressed memory. For the same setup as before, Wolfe's encoding requires a LAT storage overhead of about 3.5% over the original program's code size, when eight cache lines are represented by each LAT entry.

4. ADAPTIVE COMPRESSION OF INSTRUCTION CACHE LINES

The architecture presented by Wolfe requires the Huffman coding table be stored alongside the compressed program in memory, or alternatively a static table determined by profiling a number of 'representative' sample programs may be hard coded into the cache refill logic. The distribution of instruction set usage in embedded applications varies widely from program to program [10], which suggests that the static approach to compression might be improved upon with a semiadaptive or adaptive technique when applied to a specific application. Given the requirement of explicitly storing the compression model for semiadaptive algorithms alongside the encoded data, we now give consideration to adaptive techniques which address these issues.

4.1 LZW for cache lines

We propose replacing the Huffman coding technique employed by Wolfe's CCRP with an LZW-based scheme. Instead of storing a static or semiadaptive Huffman coding table in the decompression unit we compute at compression-time an LZW dictionary which will be appropriately initialized and adaptively maintained at run-time during the execution of the program.

As a starting point, we investigate the simple idea of compressing each cache line individually using the LZW algorithm. Before coding each cache line, the LZW dictionary contains initial entries for the alphabet $\sum$ of unique bytes 0..255. The cache line is then coded, padded to provide byte alignment, and stored in memory. As before, a LAT table is used to locate compressed code in memory. At run-time, whenever a cache miss occurs the LZW dictionary is reset, the compressed code is located in memory using the LAT, and the cache line is decompressed and placed in the instruction cache.

Results from the application of this technique are presented in Section 6. Unfortunately, except for cache lines of length 64 bytes or greater, the technique results in expansion rather than compression of the input. In the following sections we introduce a new approach to extract additional context for compressing instruction cache lines that extends beyond a single cache line boundary. A cache line will be compressed using the context obtainable from those other cache lines guaranteed to have been encountered on all of the possible run-time paths leading to an instruction cache miss for the line. To this end we will review the immediate dominance relation between basic blocks in a control flow graph from compiler design [1, 21], introduce a new data structure we term the compulsory miss tree, and finally present our improved adaptive compression technique for cache lines.

4.2 The immediate dominance relation

An executable program's object code consists of encoded machine instructions each of which can be classified as either a branch instruction or one which does not affect control flow. Control moves sequentially through a program until a branch is encountered. At such a point the processor begins executing instructions at the destination of the branch or at the address following the branch. A *basic block* is a maximal sequence of consecutive instructions in which flow of control enters at the beginning of the block and leaves at the end. We can partition the object code for a program into a set of basic blocks. The *control flow graph* (CFG) of a program is an abstract representation of the code which models, at compile-time, possible flow of control within the program at run-time[1]. Its nodes are basic blocks, and an edge $\langle x, y \rangle$ indicates that execution of block y can immediately follow that of block x in some execution sequence. The graph has a distinguished node *entry* representing the program's unique entry point. CFGs are used in many compiler and link-time optimizations, and their construction is well described in the literature [1, 21].

Let x, y be blocks in a CFG. We say x dominates y (written $x\ dom\ y$) if every path from *entry* to y in the CFG passes through x. Every node dominates itself and every node is dominated by *entry*. x strictly dominates y ($x\ sdom\ y$) if $x\ dom\ y$ and $x \neq y$. x immediately dominates y ($x\ idom\ y$) if $x\ sdom\ y$ and x does not dominate any other dominator of y. Every node other than *entry* has a unique immediate dominator. This relation may be represented by an immediate dominator tree with *entry* at the root and edges representing the *idom* relation between nodes.

[1]The term control flow graph typically refers to the representation of control flow within a single procedure. Here we use the term to mean an interprocedural control flow graph of the supergraph form [22].

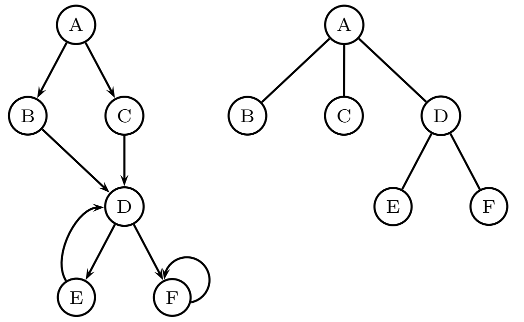

Figure 4: Example control flow graph and its immediate dominator tree

Figure 4 presents a simple control flow graph and its associated immediate dominator tree.

4.3 Compulsory miss tree

When a processor is unable to locate the data that it requires in its cache, a cache miss occurs. These misses can be characterized as either compulsory, capacity or conflict misses. Capacity and conflict misses occur as a result of the physical size of the cache and the associated block placement strategy. A compulsory miss occurs on the first access to a piece of data through the cache, that is, when the data was not previously loaded [11].

In this section we present an algorithm to construct a compulsory miss tree for a given program and instruction cache. Nodes in this tree represent individual instruction cache lines in the program which may be loaded from memory. When any type of miss occurs for a given node in the tree, all ancestors of the node will previously have caused their compulsory miss in the cache to occur.

The construction proceeds as follows: First, the boundaries between basic blocks in the object code are identified. Then we determine the boundaries in the object code between cache line data for loading into the instruction cache. Next, the CFG is constructed from the set of basic blocks, and the associated immediate dominator tree is derived. Recall that for each basic block in the immediate dominator tree, execution of all its ancestors will have preceded execution of the node itself. This will have occurred in order from the root of the tree to the node in question, but other basic blocks may also have been executed between nodes on that path. When a given basic block is encountered its instructions will execute in sequential order. Thus cache lines containing instructions for the basic block will be found in contiguous memory locations, and data from them will be requested in order from the cache.

We define two relations, *touches-cl* for basic blocks and *touches-bb* for cache lines (Figure 5), which will help us later when we relate the dominance control flow information to the order of compulsory misses in the instruction cache.

$$\textit{touches-cl}(\text{basicblock } b) = [c \mid c \in \textit{cachelines}, \textit{c.addressrange} \cap \textit{b.addressrange} \neq \emptyset]$$
$$\textit{touches-bb}(\text{cacheline } c) = [b \mid b \in \textit{basicblocks}, \textit{b.addressrange} \cap \textit{c.addressrange} \neq \emptyset]$$

Figure 5: Definition of touches-cl and touches-bb, relating basic blocks and cache lines to each-other

Given a basic block x, *touches-cl* will return a set containing those cache lines y which contain part of the data for basic block x. Similarly, given a particular cache line q, *touches-bb* will return the set of basic blocks r which contribute to the data stored in cache line q.

Together, this knowledge allows us construct the compulsory miss tree by employing the algorithm shown in Figure 6. Lines 3–8 identify the root of the compulsory miss tree, that is, the first cache line of instruction data that will miss when the program executes. This corresponds to the cache line containing the first instruction in the program's *entry* basic block (the root of the immediate dominator tree). Line 10 performs a pre-order traversal of all basic blocks b in the immediate dominator tree, determining for each the point in the compulsory miss tree where its associated cache lines cl should be attached (line 11). As each cache line cl may contain data from multiple basic blocks, its compulsory miss will occur the first time any of those blocks in touches-bb(cl) execute. For this reason, only the compulsory misses generated on paths to all basic blocks in touches-bb(cl) may be used as ancestors for cl in the compulsory miss tree, as determined by lines 15 and 16. In the event that b, the current basic block being traversed, does not contain any cache lines attached to the compulsory miss tree, the misses guaranteed to have previously occurred prior to its execution are those resulting from execution of its immediate dominator. In this situation, the immediate dominator's deepest cache line in the compulsory miss tree is used as ancestor for the first cache line in block b (lines 18 and 19).

Figure 7 shows a simple MIPS assembly language program with its associated machine-level object code. Two views of the program are provided - a basic block view (showing the source code), and a cache line view (showing the corresponding machine-level object code). We use a hypothetical instruction cache that holds 8 bytes (2 instructions) per line. The *touches-bb* and *touches-cl* relations are also indicated. The CFG and immediate dominator tree introduced in Figure 4 are those that correspond to this program. Applying the algorithm outlined in Figure 6 to the example program yields the compulsory miss tree shown in Figure 8.

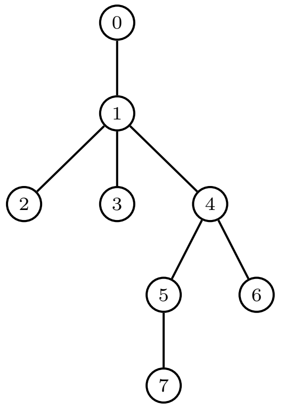

Figure 8: Compulsory miss tree

4.4 Improved LZW for cache lines

Our improved approach to dynamically building an LZW dictionary for cache line compression begins by building a compulsory miss tree for the program to be compressed. Recall that this tree has the property that whenever a miss occurs for a cache line, all of its ancestors will previously have triggered their compulsory miss.

4.4.1 Compression

Statically we allocate space in the LZW dictionary to those cache lines on the path to the largest number of other cache lines. In this way, nodes deeper in the compulsory miss tree will have a substantially larger amount of context available in the dictionary for use in their coding when compared to the naive LZW scheme introduced earlier in this paper. The nodes in the compulsory miss tree are weighted by a count of all their descendants. Then, beginning with only the alphabet of bytes $\sum = 0..255$ in the dictionary, the cache lines from the compulsory miss tree are compressed in order of their weight. Each cache line uses only those entries placed in the dictionary by its ancestors in the compulsory miss tree for its coding. In this way, each cache line inherits the context contributed to the adaptive model from the compression/decompression of its compulsory miss tree parent. Note that the ordering of compression guarantees all of a node's ancestors will be processed before the node itself, and hence the appropriate entries in the dictionary must be present. When the dictionary becomes full, no more adaptation takes place and all remaining cache lines are compressed using the entries placed in the (now full) dictionary by their ancestors. Note that each pointer, although fixed in length, cannot be used to address *arbitrary* entries in the dictionary since not all entries are guaranteed to have been initialized at the point when the decompression of a cache line occurs, hence only those entries *statically guaranteed to be initialized* may be addressed.

Referring to our example program from Figure 7, cache lines will be compressed in the order $\{0, 1, 4, 5, 2, 3, 6, 7\}$. The first byte from Line 0 will be compressed using only the initial alphabet for context, with dictionary adaptation starting at entry 256. Subsequent bytes will be processed in a similar fashion using the expanding dictionary for coding purposes. However, at the start of each new cache line, the initial context for compression comprises only those entries added to the dictionary by the line's ancestors in the compulsory miss tree. For example while node 5 will have adapted the dictionary at compression-time prior to the compression of node 2, the encoding of node 2 *must not* make use of those dictionary entries; only the entries resulting from the compression of nodes 0 and 1, as well as those resulting from processing the node itself, may be referenced when node 2 is being processed. An outline of the process is shown in Figure 9.

```
construct_compulsory_miss_tree(): CacheLine is

   BasicBlock idomTreeRoot := build_immediate_dominator_tree()
   CacheLine cmTreeRoot := first_cache_line(idomTreeRoot)

   cmTreeRoot.processed := true
   cmTreeRoot.cmTreeDepth := 0
   cmTreeRoot.cmTreeParent := nil

   for each BasicBlock b in pre_order_traverse(idomTreeRoot)
      for each CacheLine cl in touches-cl(b)

         if cl.processed = false then

            BasicBlock a := common_idom_ancestor(touches-bb(cl))
            cl.cmTreeParent := deepest_cmt_cl(a)

            if cl.cmTreeParent = nil then
               cl.cmTreeParent := deepest_cmt_cl(a.idomTreeParent)
            end -- if

            cl.cmTreeDepth := cl.cmTreeParent.cmTreeDepth + 1
            cl.processed := true

         end -- if

      end -- for each
   end -- for each

   return cmTreeRoot

deepest_cmt_cl(x)
-- Returns the cache line deepest in the compulsory miss tree from the set
-- touches-cl(x). Returns nil if no cache line in touches-cl(x) has been added
-- to the tree.

common_idom_ancestor(x)
-- Returns the deepest common ancestor of the set of basic blocks x in the
-- immediate dominator tree.

first_cache_line(x)
-- Returns the cache line with lowest address in the set touches-cl(x).

build_immediate_dominator_tree()
-- Constructs an immediate dominator tree for the input program and returns
-- its root node.
```

Figure 6: Compulsory miss tree construction algorithm

Cache line view			Basic block view		
cl#	touches-bb	obj code	source code	touches-cl	bb
0	{A}	0x8C080200	lw t0,512(zero)	{0, 1}	A
		0x8C090204	lw t1,516(zero)		
1	{A}	0x10080004	beq zero,t0,<C>		
		0x240A0000	li t2,0		
2	{B}	0x01095020	add t2,t0,t1	{2, 3}	B
		0x10000002	b 420 <D>		
3	{B, C}	0x014A5020	add t2,t2,t2		
		0x01205020	add t2,t1,zero	{3}	C
4	{D}	0x8C0B0208	lw t3,520(zero)	{4, 5}	D
		0x100A0004	beq zero,t2,<F>		
5	{D, E}	0x016A5820	add t3,t3,t2		
		0x214AFFFF	addi t2,t2,-1	{5, 6}	E
6	{E}	0x1000FFFB	b <D>		
		0xAC0B0208	sw t3,520(zero)		
7	{F}	0x1000FFFF	b <F>	{7}	F
		0x00000000	nop		

Figure 7: View of program object code showing cache lines (*cl*) and basic blocks (*bb*)

Index	Phrase
0	0x00
1	0x01
.	.
.	.
255	0xFF
256	*uninit*
257	*uninit*
.	.
.	.
2^n	*uninit*

Index	Phrase
0	0x00
.	.
.	.
255	0xFF
256	0x8C,0x08
257	0x08,0x02
258	0x02,0x00
259	0x00,0x8C
260	0x8C,0x09
261	0x90,0x02
262	0x02,0x04
263	*uninit*
.	.
.	.
2^n	*uninit*

Index	Phrase
0	0x00
.	.
.	.
255	0xFF
Line0 adaptations	
Line1 adaptations	
Line4 adaptations	
Line5 adaptations	
Line2 adaptations	
.	*uninit*
.	.
.	.
2^n	*uninit*

Figure 9: Initial dictionary; dictionary after compressing line 0; and structure of dictionary after five lines lines from the compulsory miss tree have been compressed

4.4.2 Decompression

LZW generally inserts new entries at the next available location in the dictionary during both compression (as we did above) and decompression. Since we cannot guarantee the total order in which compulsory misses will occur at run-time, we cannot employ such an approach during decompression. If we did, adaptations could occur at different locations in the dictionary to those that occurred during compression, thereby invalidating the indices used for the encoding. For example, a sample execution of our program from Figure 7 which begins by traversing basic blocks a and b will cause the compulsory miss, and hence decompression, of cache lines in the order $0, 1, 2$ and 3. At compression-time, however, the lines were compressed and modified the dictionary in the order $0, 1, 4$.... Clearly if we allow the decompression at run-time adapt the dictionary from the next available location, we will not reconstruct the same model used at compression-time, resulting in serious errors in the 'decompressed' code.

For this reason, encoded cache lines require some additional bookkeeping information for use by the decompression unit to ensure fidelity of the dictionary model. They need to indicate whether or not their decompression should adapt the dictionary, and if so where entries in the dictionary should be placed as the node is decompressed. A single bit is stored to indicate whether processing a node should lead to a modification of the dictionary, and if so the following index will identify the location in the dictionary where the first new entry should be placed. This information is readily available at compression-time, and is output as part of the compressed cache line encoding. When a miss occurs in the instruction cache at run-time, its compressed form is located in memory. If the first bit is a zero, then it shows that all codes needed for its decoding have already been initialized and this node adds no new entries to the dictionary as it is expanded. However, if the first bit is set to one, then the regular LZW decompression technique is applied, adapting from the location identified by the first index of the cache line's encoding. For an example of the encoding, see Figure 10, where the compressed version of line 0 from our example is presented.

Our LAT tables are slightly larger than Wolfe's, since we allow for the expansion of data where they reject it. We need to do this since that is effectively how LZW attains compression: initial data coding causes expansion, but also provides those dictionary entries that later allow us obtain good compression ratios. The maximum size of a compressed cache line using our scheme includes the overhead when a cache line contributes to the LZW dictionary (1 bit flag plus the index used to address the location to adapt) plus the worst case encoded size when no compression occurs and every byte is replaced by an index to the LZW dictionary (requiring $x \cdot n$-bit indices for a dictionary using indices of length n bits, in a cache line of x bytes).

5. RELATED WORK

Work on code size reduction has focused on two general approaches termed code compaction and code compression. Code compaction applies transformations to the input program creating an output program which is semantically identical to the input, but which requires less space for storage. Binaries resulting from code compaction are directly executable on their target architectures and no decompression step is required. The transformations employed for code compaction include traditional compiler techniques such as redundant and dead code elimination applied in an aggressive interprocedural fashion, combined with procedural abstraction and cross jumping [4, 7, 9].

Code compression techniques on the other hand recode the input binary in a form which is *not* directly executable on the target processor, requiring an intermediate decompression step before execution can occur. Such techniques are based on coding and information theory concepts. They are orthogonal to the code compaction methods mentioned above. The techniques have been applied at various levels of granularity including an entire program's abstract syntax tree [8], individual procedures [5, 6], basic blocks [18], instruction sequences which do not extend beyond a basic block [14], individual cache lines [12, 25] and indeed individual instructions [27].

A number of semiadaptive statistical techniques have been proposed for fine grained code compression, such as Xie's *variable to fixed coding scheme* [26] and Lekatsas' *semiadaptive markov compression* (SAMC) [16]. The compulsory miss tree based approach described in this paper does not solve the problem of maintaining a statistical model adaptively, and so statistical techniques are not described in this section.

An obvious semiadaptive approach to code compression [27] takes a list of all instructions used in a given program's object code and extracts a list of the unique binary encodings to form a dictionary. The program is then recoded, replacing each instruction with a pointer to its original encoding. Decoding is implemented on-chip via a dictionary lookup. Locating compressed code in memory is straightforward since both the uncompressed program and its compressed version use an identical memory address space. Thus decoding can begin at any instruction in the stream and is not restricted to start at branch targets.

Lefurgy [14] introduces a semiadaptive dictionary-based compression scheme where code words are expanded into a sequence of instructions during the processor decode cycle. The code words are prefixed with unused opcode encodings in the base instruction set, and interspersed with uncompressed instructions. The solution used to locate compressed code in memory involves rewriting branch instructions and

Adapt LZW dictionary (binary$_2$)	Adaptation location (decimal$_{10}$)	LZW coded cache line data (hexadecimal$_{16}$)							
1	256	0x8C	0x08	0x02	0x00	0x8C	0x09	0x02	0x04
Encoded using a single bit	Each dictionary index is encoded using n bits (n = LZW pointer length)								

Figure 10: Encoding format for Line 0

program jump tables containing destination addresses of control flow. These are updated to point to the location of code in the *compressed* instruction memory. The scheme enables decoding from basic block boundaries by allowing dictionary code words to replace only instruction sequences which do not overlap basic blocks.

The only attempt at applying adaptive techniques to code compression that we are aware of was presented by Lin [18], where LZW was applied to individual basic blocks in VLIW code. The average basic block size in their experiments was reported at 454 bytes and this provided sufficient context for LZW to achieve compression ratios of 83-87%, and a variant of LZW 75%. To locate code in memory they used a lookup table which mapped from uncompressed branch destinations to corresponding locations in compressed memory. This did not adversely affect the compressed code size since on average there were just 80 basic blocks in the evaluated programs. To enable basic block decoding at branch targets the LZW dictionary is reset before encoding/decoding each block. However such a distribution of few but large basic blocks is uncharacteristic of normal RISC code that has been optimized for size, and hence their approach is not applicable to a large class of embedded systems.

6. RESULTS

To evaluate our new approach to code compression, eight benchmarks from the MiBench [10] and MediaBench [13] suites were selected. These were compiled for the MIPS architecture using GCC 3.3.1 with size optimization enabled (-Os) and then compacted using version 0.3 of the Diablo link-time optimizer for MIPS [20]. We began by reproducing the results presented by Madou [20], then we extended Diablo to output data about the basic block boundaries in the optimized binaries along with information about their associated interprocedural control flow graphs. The object code and associated control flow graph were used as input to our compression algorithm which then produced the results presented here. It should be noted that previous object code compression schemes working at the granularity of cache lines have not applied link-time optimization and compaction prior to obtaining their results, thus leaving more redundancy in the benchmark code upon which they applied their techniques. We present results only in terms of code size reduction, leaving an implementation and evaluation of the performance overhead incurred in employing the technique to future work.

We experimented with varying both the dictionary size/index length (from 9 to 12 bits) and cache line size (ranging from 16 to 128 bytes). For comparison, data for the basic scheme described in Section 4.1 where LZW is applied to each cache line independently is also presented. Using its assigned virtual address the first instruction of the object code did not always align to the start of a cache line boundary, nor did the code necessarily fill the final cache line in which it was contained; in these situations, the cache lines were padded with zeros until they became correctly aligned. We

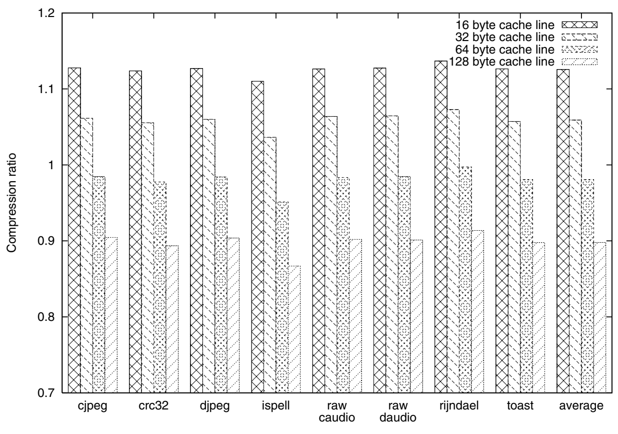

Figure 12: Compression ratio for regular LZW, applied to cache lines of 16 to 128 bytes

padded all compressed cache line encodings to byte alignment. Our results account for all overhead involved in the scheme, including the LAT tables (requiring a constant overhead of 6.25%, 3.515%, 1.953125% and 1.072419% of the original program code size for cache lines of length 16, 32, 64 and 128 bytes respectively, using dictionary indices of size ranging from 9 to 15 bits per index and eight cache lines per LAT entry).

Figures 12 and 13 present the compression ratios (compressed size/uncompressed size) achieved in our experiments for all benchmarks, using various length cache line sizes with 9 bit LZW dictionary indices. A compression ratio of 1 indicates no size reduction, greater than 1 expansion, and less than 1 compression. Figure 11 shows the original code size (when padded to cache line alignment), and that resulting from applying the two approaches for each cache line length (results in figure 11 represent our raw results, from which the compression ratios were derived. They exclude the LAT overhead, though it is fully accounted for in all other summary descriptions and results). On average, 7.7%, 11.9%, 17.7% and 18.3% of the original code size is eliminated using our compulsory miss tree based technique for cache lines of length 16, 32, 64 and 128 bytes respectively, compared with reductions of -12.6%, -5.9%, 2.0% and 10.2% for regular LZW. We do not present results for larger dictionaries since none of them showed an improvement over those just described. We rationalize this somewhat unexpected behavior as follows:

The control flow graph presented by Diablo is conservative, resulting in a shallow dominator tree. The CFG reconstruction algorithm employed by Diablo assumes that indirect branches may pass to any relocation target in the object code. This impacts the compression attainable using our scheme since increasing the dictionary size cannot provide additional context for nodes near the top of the compul-

	Original size				LZW size				Improved LZW size			
	16	32	64	128	16	32	64	128	16	32	64	128
CRC 32	23328	23360	23424	23424	24756	23837	22440	20678	20268	19368	18304	18789
CJPEG	98240	98272	98304	98304	104638	100851	94876	87868	84568	81393	78317	77370
DJPEG	118752	118784	118848	118912	126385	121738	114648	106192	105792	102057	96520	95331
ISPELL	86624	86656	86720	86784	90751	86760	80800	74298	73856	69667	66533	66983
RAW-CAUDIO	13808	13824	13888	13952	14688	14219	13387	12435	12057	11685	10969	11003
RAW-DAUDIO	14064	14080	14144	14208	14977	14494	13653	12651	12135	11643	11131	11301
RIJNDAEL	39856	39872	39936	39936	42811	41374	39053	36060	36579	35691	34219	35398
TOAST	61168	61184	61248	61312	65074	62526	58858	54381	55061	53643	51243	50263

Figure 11: Original code size and that resulting from compression (in bytes), excluding LAT overhead, for a variety of cache line sizes

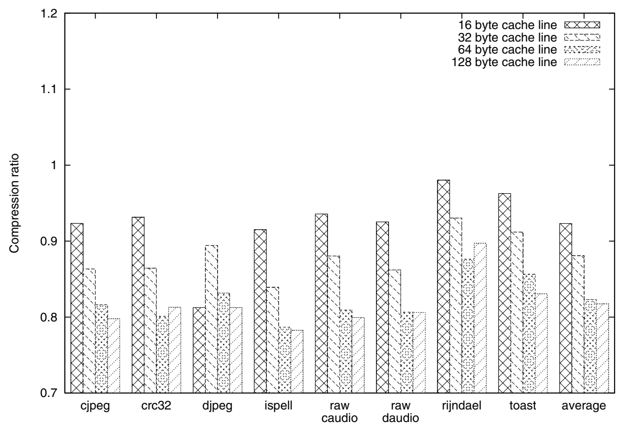

Figure 13: Compression ratio for compulsory miss tree based LZW, applied to cache lines of 16 to 128 bytes

sory miss tree, the most common location for a line during our experiments (see for example the immediate dominator tree and associated compulsory miss tree for the CRC32 benchmark in Figures 14 and 15 respectively). A tighter integration with the code generation tool chain would provide less conservative information, giving rise to a deeper immediate dominator tree, resulting in a deeper compulsory miss tree.

The compression ratio results we present here are not directly comparable to those published elsewhere in other research on code size reduction techniques. The difficulty with comparing results published in the literature is the diversity of the experimental setups employed, as has previously been noted [2]. Unfortunately, given the extensive literature on code size reduction techniques [23], it is impractical to reimplement existing methods to provide a fair comparison each time a new technique is developed.

Consider, for example, that the schemes we reference in this paper have targeted the Compaq Alpha [5, 6, 7], MIPS [20, 25], A RISC like abstract machine [4], ARM [14, 27], Texas Instruments TMS320C6x [18, 26], PowerPC [12, 18], i386 [18] and PDP 11 [9]. Each of these machines employ a different Instruction Set Architecture, with a distinct encoding, providing different opportunities for compression. The benchmarks evaluated vary from one investigation to another, with some combination of programs from SPEC '95, SPEC '92, MiBench, MediaBench and vendor supplied code in the mix. Preparation of the input binaries employed a spread of optimizations ranging from none at all, through 'normal' optimization levels, to size optimization. Indeed some papers do not discuss the preparation of the binaries they compress in any detail. Taking into account the fact that different compilers were used (various versions of GCC, the massively scalar compiler, and vendor supplied compilers), it is obvious that the results obtained across papers are simply not amenable to direct comparison. Furthermore some of the techniques described compress statically linked programs, while others do not include any library code in their results. Indeed while some researchers have applied program compaction prior to compression as we do, the vast majority have not.

7. FUTURE WORK

Our heuristic statically allocated space in the LZW dictionary to those nodes with the largest number of children in the compulsory miss tree. The intuition behind this allocation was to maximize the number of dictionary entries available to as many cache lines as possible. Other heuristics might be investigated giving the potential for better results. For example, it would be possible to modify our approach and reserve a single section of the dictionary for use by all cache lines which were not statically allocated space in the dictionary. As these nodes do not provide context to any other node in the compulsory miss tree, they may share a single section of the dictionary which is cleared and adapted as their compression/decompression takes place. Indeed suitable control flow analysis might identify collections of cache lines which can share sections of the dictionary, effectively increasing its size and the amount of context available for coding of certain cache lines.

Applying the standard intraprocedural approach to computing the dominance relation gives conservative results if applied to a control flow graph representing an entire program. It always gives rise to a tree structure, the immediate dominator tree, when the relation is transitively reduced. If a more precise analysis is applied to CFGs of the supergraph variety, considering only feasible paths which may actually be traversed at run-time, the transitive reduction of the dominance relation may give rise to a directed acyclic graph (DAG). This is because the intraprocedural computation of dominance considers all paths through the graph traversable while the representation of procedure calls/returns in a CFG introduces a flow-sensitivity of paths which is simply ignored. With a more careful analysis of control flow, additional information about which basic blocks are guaranteed to be encountered on all paths to a given block may be extracted. A full investigation of flow-sensitive interprocedural dominance, and the construction of an associated compulsory miss DAG is the subject of future work.

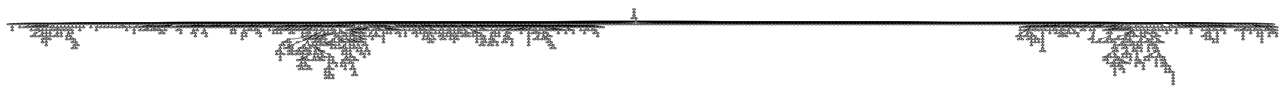

Figure 14: Immediate dominator tree for CRC32 benchmark

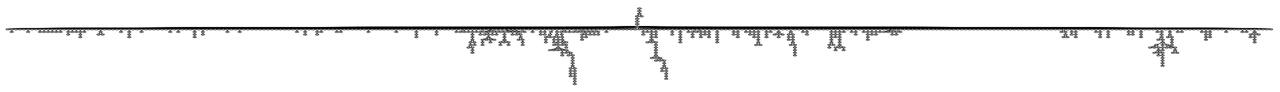

Figure 15: Compulsory miss tree for CRC32 benchmark, based on a 32 byte cache line size

This analysis would allow certain cache lines use more of the constructed LZW dictionary than is currently the case.

In this study we investigated the use of the LZW compression algorithm, one of the LZ78-based adaptive dictionary compression techniques. Future work will look at employing other popular adaptive schemes, including those grouped under the LZ77 class of algorithms. We believe that by applying appropriate constraints these techniques can be efficiently implemented and parallelized in hardware for object code compression. Indeed, specialized adaptive algorithms exploiting the nature of object code also merit closer study.

8. CONCLUSIONS

In this paper we have presented a new approach to object code compression employing an algorithm from the class of adaptive schemes previously dismissed by many researchers. Our approach makes use of knowledge about the partial order in which compulsory misses for a program will occur in an instruction cache at run-time, summarized in a data structure we term the compulsory miss tree. This tree is used as a basis to appropriately construct an adaptive dictionary-based model for compressing/decompressing individual instruction cache lines. We have demonstrated that adaptive object code compression at this granularity can yield results sufficient to warrant further investigation. Furthermore, we have described a number of avenues for future research in this area. The remaining question regarding the use of adaptive algorithms for object code compression is whether such schemes can be implemented with a sufficiently high decode efficiency to make the overall approach practical. With appropriate constraints applied, we believe the answer to this question will be yes.

Acknowledgments

We are grateful to Edsko de Vries for helpful discussions and for critical feedback on early drafts of this paper. The work of John Gilbert was supported by a Trinity College Dublin Postgraduate Award.

9. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] 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.

[3] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression.* Prentice Hall, 1990.

[4] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 139–149, New York, NY, USA, 1999. ACM Press.

[5] Saumya Debray and William Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, New York, NY, USA, 2002. ACM Press.

[6] Saumya Debray and William S. Evans. Cold code decompression at runtime. *Commun. ACM*, 46(8):54–60, 2003.

[7] 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.

[8] Michael Franz and Thomas Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[9] Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analyzing and compressing assembly code. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 117–121, New York, NY, USA, 1984. ACM Press.

[10] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2002.

[12] T.M. Kemp, R.K. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6):807–812, 1998.

[13] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30*, pages 330–335. IEEE Computer Society, 1997.

[14] C. Lefurgy, P. Bird, I-Cheng Chen, and T. Mudge. Improving code density using compression techniques. In *MICRO 30*, pages 194–203. IEEE Computer Society, 1997.

[15] Charles Robert Lefurgy. *Efficient Execution of Compressed Programs.* PhD thesis, Department of

Computer Science and Engineering, University of Michigan, 2000.

[16] Haris Lekatsas. *Code Compression For Embedded Systems*. PhD thesis, Department of Electronic Engineering, Princeton University, 2000.

[17] Stan Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 4(1):12–38, 1999.

[18] C. Hong Lin, Y. Xie, and W. Wolf. LZW-based code compression for VLIW embedded systems. In *DATE '04*, page 30076. IEEE Computer Society, 2004.

[19] Steven Lucco. Split-stream dictionary program compression. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 27–34, New York, NY, USA, 2000. ACM Press.

[20] Mathias Madou, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. Link-time compaction of MIPS programs. In *Proceedings of the International Conference on Embedded Systems and Applications*, pages 70–75, Las Vegas, 6 2004. CSREA Press.

[21] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[22] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *POPL '81*, pages 219–230. ACM Press, 1981.

[23] Rik van de Wiel and Mario Latendresse. The code compression bibliography, 2004. www.iro.umontreal.ca/~latendre/codeCompression/.

[24] Terry A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[25] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded RISC architecture. In *MICRO 25*, pages 81–91. IEEE Computer Society Press, 1992.

[26] Yuan Xie, Wayne Wolfe, and Haris Lekatsas. Code compression using variable-to-fixed coding based on arithmetic coding. In *Proceedings of the Data Compression Conference*. IEEE Computer Society, 2003.

[27] Y. Yoshida, Bao-Yu Song, H. Okuhata, T. Onoye, and I. Shirakawa. An object code compression approach to embedded processors. In *ISLPED '97*, pages 265–268. ACM Press, 1997.