# An Architecture-Oriented Approach to Code Compression for Embedded Processors

Cheng-Hong Li Steven M. Nowick
Department of Computer Science
Columbia University
New York, NY 10027
{cheli,nowick}@cs.columbia.edu

## ABSTRACT

This paper presents an efficient and general framework for code compression for embedded systems. The method is based on simple Huffman encoding, but where limited yet powerful higher-order correlations are captured and exploited by using architecture-oriented models. The proposed symbol formation, modeling and encoding steps together achieve high-quality code compression, yet the decoding procedure is straightforward, and can be implemented by a simple decoder. To avoid modifications of microprocessor cores, the decoder is placed between the instruction memory and cache, and is triggered only on cache refills.

Applying our framework to ARM benchmark programs, including from MediaBench, the best code compression ratio achieved is $48.1\%$, and the best overall ratio achieved is $56.7\%$ (including decoder hardware overheads). The average overall ratio is $59.4\%$. We believe this work is the first to exploit a wide range of detailed architectural information, under a higher-order statistical model, in an effective and practical code compression scheme for embedded systems.

## 1. INTRODUCTION

Embedded systems usually have limited storage space due to cost, system size, and power constraints. In many designs, programs are stored in non-volatile circuitry, like ROMs. The size of the programs usually has a direct impact on system cost. As embedded systems are deployed in more versatile environments, the functional requirements of embedded applications grow more complex. As a result, program size becomes an ever-increasing concern to system designers.

Several efforts have been proposed to alleviate the code-size problem. Architects have proposed denser instruction sets to facilitate reduced code size [1, 26]. Compiler optimizations are proposed to generate smaller executables. Another approach is to apply traditional data compression techniques to reduce program size, using existing instruction set architectures.

In this paper we present an efficient and general compression scheme to reduce code size for embedded systems. The compression is a post-compilation process. The decompression is performed by a simple and fast hardware decoder when the program is executed. No modifications to the compiler or to the processor core are needed. The scheme is specifically targeted toward ARM microprocessors, yet the framework can be easily adapted to other architectures. Our focus is on ARM because it is both the leading embedded architecture, and at the same time the most challenging for code compression: having CISC-like code density in a sophisticated RISC architecture [26].[1]

The compression method is based on simple Huffman encoding, but where limited yet powerful higher-order correlations are captured and exploited to improve compression efficiency. The methods to capture high-order correlations are architecture oriented. Different instruction types, and their various submodes, are first used to define variable-length symbol sizes and formats. The symbols are then sorted into different sets based on the intra-instruction context (i.e. current instruction type and operation submode) and state (i.e. position in the instruction). This step is called *modeling*. Each set of symbols, corresponding to a unique context and state, is then encoded separately. Two approaches to Huffman encoding are used: standard Huffman coding, and a new heuristic "curve-fitting" approach which approximates the former but with reduced decoder overheads. The proposed symbol formation, modeling and encoding steps together achieve high-quality code compression, yet the decoding procedure is straightforward, and can be implemented by a simple decoder.

Applying our framework to a number of ARM benchmark programs, including a number from the popular MediaBench suite [15], the best code compression ratio achieved is $48.1\%$, and the best overall ratio achieved is $56.7\%$ (including decoder hardware overheads). The average overall ratio is $59.4\%$. These compression results are the best that have been reported for ARM using a scheme which requires no modification to the compiler or processor core (see the end of Section 9).

These results are achieved through a number of contributions: (1) Higher-order entropies [5] are systematically exploited, in a limited but targeted way, to obtain improved code compression. In particular, instruction types and submodes, as well as symbol position within an instruction, are used, respectively, as "architecture-oriented' context and state, in order to capture subtle and more nuanced correlations within the actual machine program sequence, which have not been fully explored in previous approaches. (2) An architecture-based symbol formation is proposed. The symbols we consider include simple fixed-length bytes and halfwords (which often match instruction field boundaries), as well as variable-length *field-oriented* symbols, including both contiguous and non-contiguous bit patterns, which correspond to fields in the original ARM instruction formats. (3) A new approximate Huffman-based coding technique is proposed, called "curve fitting". In this coding approach, multiple distinct sets of initial (uncompressed) symbols are mapped onto a *single* predefined set of (compressed) Huffman codewords, using an optimal "fitting" scheme. The resulting code compression ratio in practice is only slightly degraded, yet the hardware decoder is greatly simplified. In particular, only a single "tokenizer" is required, shared by many distinct symbol sets, to identify compressed Huffman codes. (4) A simple and extensible hardware decoder design is proposed. We believe this new compression framework and decoder architecture can be easily expanded to encompass further refinements to the encoding scheme, in terms of extending the symbol sets and identifying more complex higher-order correlations.

## 2. BACKGROUND

This section begins with some brief background on the ARM instruction set. Then Huffman encoding is reviewed. Finally, a brief taxonomy of code compression techniques is also presented.

**ARM Architecture.** The ARM architecture is a RISC architecture widely used in embedded systems. A key strength of ARM compared to other RISC instruction sets is its inherent CISC-like code density [26]. Its instruction set [1] defines 32-bit fixed-length instructions, a selection of whose formats is shown in Fig. 1. Type (a) instructions are general *data processing* instructions. The exact format of the lower 12 bits is pivoted on the $I$ bit (bit 25). Type (b) instructions are *single data transfers*. Bit 20 indicates whether it is a `ldr` (i.e. load) or `str` (i.e. store) instruction. Type (c) instructions are *branches*. The L bit flags whether the link register will be updated after the branch is taken. In general, in all the above instruction formats, the field from bit position 27 to 25 specifies the "instruction type", as highlighted in the figure, while the field from bit position 24 to 20 specifies the "instruction sub-mode" (e.g., providing further details of the operation). For

[1] In contrast, a number of other RISC architectures have much worse code density than ARM, such as the MIPS ISA.

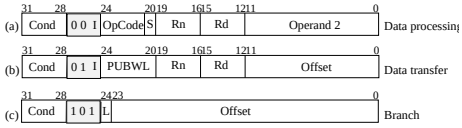


**Figure 1: ARM instruction formats.**

simplicity, the type and operation sub-modes of an instruction together are hereafter called *instruction modes*, and each mode can be identified by the *instruction mode field* from bit position 27 to 20.

**Taxonomy.** Code compression methods can be classified by several parameters, including their fundamental symbol encoding techniques, how they are adjusted towards different application programs, what the atomic compression units are (called *symbols*), and the architecture of their hardware decoders. These aspects are briefly explored below. A more detailed comparison to recent related work will appear in Section 9.

Traditional encoding techniques fall into two groups: *statistical* and *dictionary* encoding. Statistical encoding, such as Huffman encoding [11], assign shorter codewords to more frequent symbols. This approach has been widely used in code compression [27, 14, 19, 20, 13]. It relies on *skewed* frequency distributions (i.e. with lower entropy) to achieve better compressions. There are two ways to make symbol frequency distributions more amenable to compression. One is by merging shorter input units with high correlations to form new symbols, so the entropy of the new symbols is reduced. This approach is called blocking, or *symbol formation* in this paper. The other approach is to consider *modeling*, where symbols are sorted into various symbol sets based on their previous symbols in symbol sequence (called *context*), or where they are sorted into symbol sets based on the actual position of a symbol in the symbol stream, called *state* [5]. This paper presents a Huffman-based code compression scheme. Various symbol formation and new models are also proposed.

A compression scheme can also be categorized by how it adjusts itself to an input program. Most code compression schemes adopt *semi-adaptive* compression method, where a compression scheme is optimized for each input program [21, 28, 16, 8, 9, 25, 19, 20, 4, 10, 3, 24, 23, 19, 18]. The semi-adaptive approach assumes that application programs are usually known during the embedded system design process, and the compression scheme can be optimized for each program in advance. In this paper, a semi-adaptive scheme is adopted.

An orthogonal classification is to consider the characteristics of symbols. In code compression applications which deal with actual machine instructions, *fixed-length* symbol sets, such as use of bytes or halfwords, are simple to use but may be suboptimal for compression. A *variable-length* symbol set is somewhat more complex to handle, in both compression and decompression, but is beneficial to compression, because the symbol lengths and formats can be customized to the fields of the machine instruction formats, and thus provide a better match to the inherent instruction syntax. Therefore, they can better capture the true semantics of an embedded machine program. Both types of symbol sets are considered in this paper in the context of symbol formations.

Finally, in code compression the decoding is performed by a hardware decoder. Two different architectures have been proposed. In the first architecture, called *decompress on cache refill* (DCR, named by Benini et al. [9]), a decoder is placed between the instruction cache and the main memory. The decoding is activated by cache misses. This approach has been used in several recent code compression schemes [27, 14, 13, 19]. The alternative is to place the decoder between the microprocessor and the cache, called *decompress on fetch* (DF, also named by Benini).[2] In this approach, each instruction fetch initiates a decoding process. Compared to the DF architecture, the DCR architecture has several advantages. First, processor cores and the compiler do not need to be modified. Second, the performance impact due to decompression is amortized over cache misses, only occurs when there is a cache miss, and is therefore less critical. Therefore, a decompress on cache refill architecture is used in this work.

# 3. OVERVIEW

[2]The DCR style is also called a *pre-cache* architecture and the DF style is also called a *post-cache* architecture [18].

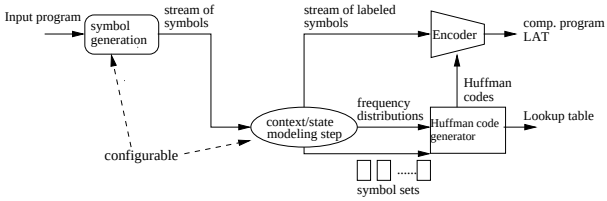


**Figure 2: The compression procedure.**

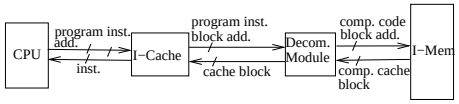


**Figure 3: Top-level organization of the hardware decoder.**

In this section, an overview of our new compression and decompression scheme is presented. Further details of our approach are presented in Sections 4 through 7.

**Compression.** Code compression is an off-line process. Its input is a compiled machine program, and its output is a compressed program and a lookup table for decoding.

Our compression scheme consists of four key steps. These four steps, which are shown in Fig. 2, are symbol generation, context/state modeling, Huffman-based code generation, and encoding. At the beginning of compression, the input program is first converted to a stream of symbols, in the *symbol generation* step. Next, the symbols are sorted into symbol sets in the *modeling step*. The sorting is based on the symbol's context or its state (i.e. position in the stream). The symbols are also labeled with tags indicating into which set each is placed. Finally, the frequency distributions of the resulting symbol sets are used by the *Huffman code generator* to construct the corresponding Huffman codes, one for each symbol set. Finally, the actual symbols are encoded by the *encoder*.

The storage of a compressed program is cache- or (memory-) block oriented, as in [27]. Each compressed cache block is stored at a byte boundary in the main memory (i.e. byte-aligned), as a compressed memory block. In this paper a 32-byte cache block is assumed [1], though this parameter can be changed according to a processor's actual cache organization. A lookup table is also generated in the Huffman-based code construction step. The structure of the lookup table is further explained in Section 7.

**Decompression.** Code decompression is performed on-the-fly by a hardware decompression module as the program is dynamically executed. This module is placed between the instruction memory and the I-Cache, as shown in Fig. 3, and is therefore a DCR architecture (i.e. pre-cache [18]). The decompression is cache- (or memory-) block oriented. On a cache miss, when a cache refill request arrives, the compressed block is first located in the memory. The block is then fetched and decompressed in order, one symbol at a time. Finally, the entire uncompressed memory block is placed in the instruction cache.

Note that, when executing a compressed program, the processor and the instruction cache are operating in the address space of the original program, which does not correspond to *actual* locations of compressed code in physical memory. Therefore, to locate a compressed cache block, the program instruction block address (i.e. start location of the memory block, in the running program's address space) must be mapped to the actual compressed code instruction block address (i.e. start of the compressed block in physical memory). This translation is made possible by using a *Line Address Translation* (LAT) table [27], as shown in Fig. 4. The LAT is a complete lookup table, mapping the start address of each uncompressed memory block to the actual location of its corresponding compressed block in memory. (It is somewhat analogous to a page table in memory management.) The LAT, like the compressed program, is also stored in the instruction memory, as shown in the figure. To speedup lookup, a *Cache Look Aside buffer* (CLB) has been proposed, to allow fast translation of recent addresses. The CLB is somewhat analogous to a TLB in memory management: it is a small fully-associative cache of recently accessed memory block addresses. The complete architecture of the decoder along with LAT/CLB is shown in Fig. 4. The design of the original LAT/CLB in [27], which was targeted to MIPS (having 24-bit instruction addresses), is slightly modified here to accommodate ARM's 32-bit instruction addresses.

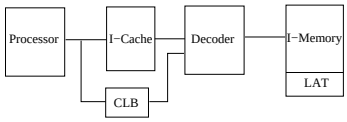


**Figure 4: Block view with LAT/CLB of the hardware decoder.**

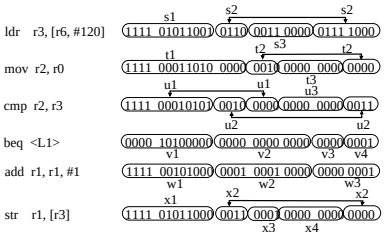


**Figure 5: Field-oriented symbol formation examples.**

# 4. SYMBOL FORMATION

The first task of designing a compression scheme is to decide what symbols are. The effectiveness of the compression and the decoding overheads are affected by several aspects of symbols. In general, better compression can be achieved by using larger symbols, as indicated in Section 2. However, a larger lookup table is needed to accommodate enlarged symbols. The area and the speed of a hardware decoder are both increased as the table becomes bigger. In this section, various symbol types are proposed and trade-offs are evaluated.

Several symbol types are considered, including bytes, halfwords, and in particular, *field-oriented* symbols composed of one or more machine instruction fields. All of the symbols are formed within a single instruction, i.e. no instruction boundaries are crossed by symbols. A field-oriented symbol set contains symbols with varied length, where each is composed of one or more instruction fields (e.g., opcode, register specifiers, sub-mode indicators, etc.). The component fields can be non-adjacent. Fig. 5 illustrates some of the proposed field-oriented symbols, in a short ARM instruction sequence.

Once a set of symbol options is defined (i.e. *symbol formation*), the next step is to determine which actual combinations of symbol types should be bound for each instruction. This binding step is called *symbol configuration.* Many possible configurations of symbol types can be selected to cover different instructions, and different experimental runs can be made before finalizing the configuration for the compression scheme of a given embedded system (see Section 8). Fig. 2 shows the configuration process as it is applied during the encoder step.

In particular, for an instruction, there are many possible ways to divide the instruction into field-oriented symbols. However, for correctness, certain rules must be followed. First, the generated symbols must cover the entire instruction, otherwise the compression loses information. Although symbols can overlap (i.e. share a certain part of the instruction), this may result in suboptimal compression and thus is not considered in this paper. Further, if one allows several different symbol lengths and formats at any given point in the instruction stream, the actual size and format of the symbol must be fully determined by its predecessor symbols or itself, otherwise the decoder will not know how to translate the corresponding codeword into the final layout of the symbol until subsequent codewords are decoded. The division of any type of instructions into symbols must be deterministic. It is an agreement between the compression and decompression processes.

In this paper, some additional restrictions to the field-oriented symbol formations are imposed. In particular, for a given instruction, the sizes and formats of symbols are decided by the *instruction mode.* Instructions are grouped into a number of classes according to their modes (i.e. instruction type and sub-mode). For each class of instructions, a deterministic way to tokenize instructions to symbols is defined. Because instruction formats and the usage of fields differ by instruction modes, the dependencies between fields can be better captured in this way. The instruction classes are not only used to decide symbols, but also later used as context in the modeling step (see Section 5). Although it is not necessary to use the same classifications for symbol formations as for context, in this paper they are unified. Nonetheless, the symbol configuration and modeling steps are truly independent. Finally, for correct decompression, the instruction mode field (i.e. bits 27-20) must be fully contained in the first symbol of an instruction. Thus, to a decoder, the sizes and formats of all the following symbols in the instruction can be fully determined after decoding its first symbol.

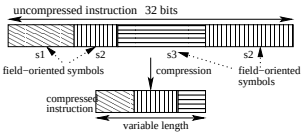


**Figure 6: Instruction compression using field-oriented symbols.**

**Example.** Several field-oriented symbol configurations are shown in Fig. 5. A sequence of instructions is shown in both assembly and binary formats. A symbol is represented by an oval circling certain portions of an instruction. For example, s1 is the first symbol of the `ldr` instruction. The `ldr` instruction is decomposed into three symbols. The first symbol consists of the most significant 12 bits, including the instruction mode field. The base register and the offset tends to have high affinity, so the second symbol merges the base and the lower 8 bits of the offset, which are sufficient to represent most offset values. The third symbol combines all of the remaining fields. The `mov` instruction is tokenized to three symbols. Its first symbol covers the first halfword, including its first operand register. Because the ARM `mov` instruction does not use this register, combining the register and the instruction mode captures this dependency. The following `cmp` (i.e. compare) is broken into three symbols. The first symbol is a halfword, merging the most significant 12 bits and the destination register, because the `cmp` does not write its result. The remaining configurations shown in the figure follow similar considerations. Note that several of the instructions include non-contiguous symbols, which span multiple non-adjacent instruction fields (e.g. ldr's s2 symbol). The general instruction compression concept using field-oriented symbols is shown in Fig. 6.

All of the selected symbol configurations that are used for the experimental results of Section 8 are shown in the above example. These configurations tended to result in the best combination of compression and table size. In principle, when two fields have high correlations, they have been merged into a single symbol, unless the merged symbol is too large. In each configuration, an instruction is tokenized into either three or four distinct symbols. The size of a symbol ranges from a nibble to a halfword.

Certain types or modes of instructions not shown here also share use one of the six symbol configurations shown in Fig. 5. For example, the miscellaneous instructions are tokenized in the same way as `ldr` instructions. Further details are explained in Section 8, where the experimental results of field-oriented symbol formation are also presented. In addition to field-oriented symbols, byte and halfword-byte mixed symbols are also evaluated. In halfword-byte mixed symbol formations, an instruction is tokenized to three symbols: the first halfword, and its following two bytes. Experimental results with byte and halfword-byte symbol configurations are also discussed in Section 8.

# 5. MODEL DESIGN

After the symbols are generated, they are assigned to different symbol sets according to a selected model. Several different models can be used; the set of modeling options must first be defined by *model design*. There are two types of higher-order correlations which can be exploited: *context* and *state*. Context considers the preceding symbols in the instruction stream. The second type of correlation which is exploited is *state:* the symbol's position in an instruction is used to define its model. Given these two modeling types, several combinations or *model configurations* can be evaluated in experiments, and the best configuration (i.e. choice of context and state) can then be selected for the encoding step of the embedded system.

In designing the models, there are several parameters that may have a significant impact on the effectiveness of the resulting compression scheme: the number of symbol sets formed, the resulting frequency distribution of each symbol set, and the number of symbols assigned to each symbol set, and so on. In designing a good model, the aim is to use as few symbol sets as possible (for smaller lookup table size), skew the distributions in each symbol set (for better compression), and choose symbol sets so that their cardinalities are small (for smaller overall lookup table size). Note that the first two requirements are

most often conflicting ones, since, by using fewer symbol sets, worse frequency distributions may result. Good compromises must be made and verified by experimental results.

**Context Model: using instruction type and mode.** In a context model, the symbols are arranged into symbol sets based on a number of previous symbols, called *context* [5]. Context captures a form of higher-order correlation between a symbol and its predecessors in a symbol stream. Hence, by using context, better compression can be obtained.

In this paper, a simplified context is used. The context is derived from the *instruction modes*. In particular, the first symbol in each instruction defines the context for all remaining symbols in that instruction. Three possible options for architecture-oriented context are proposed: *(i) itype,* where the instruction type defines the context (4 contexts); *(ii) imode,* where the instruction's detailed sub-modes are used, and each possible sub-mode forms a different context (13 contexts); and finally *(iii) himode* (i.e. hybrid imode), which falls between itype and imode in granularity, and collapses together several of the sub-modes into the same context (7 contexts).

Note that the first symbol in each instruction is treated as having no context. It carries the instruction mode information to be used by the subsequent symbols in the instruction, hence the first symbol of every instruction is always assigned to the same symbol set. However, once the context of the instruction is determined by the first symbol, all remaining symbols in the instruction are then deposited into a distinct symbol set, dedicated to this context. The reason that the first symbol is not arranged to the same symbol set as its subsequent symbols is for correct decompression. The context of a symbol must be known to successfully decode the symbol. Thus the first symbol of an instruction carrying the context cannot be distinguished by the context provided by itself, and hence, the first symbol is treated as having no context.

In more detail, the three proposed contexts are formed as follows. The *itype* context defines four classes of ARM instructions, loosely correspond to *data processing*, *single data transfer*, *branch*, and *miscellaneous* instructions. In the *imode* scheme, thirteen classes are defined. The data processing instructions are divided into six groups, depending on three pivots: whether I bit is set, the first operand is ignored or not (ex: `mov` and `mvn`) , and if the destination register is not written (ex: `tst`, `teq`, `cmp`, `cmn`). Note that two of the eight combinations are not legal in ARM: ignoring the first operand and not writing to the destination register, regardless of the I bit value. For branches, four classes are defined: both forward and backward branches are distinguished as distinct contexts, and the contexts are further distinguished depending on whether the branch modifies the Link register. For data transfer instructions, there are two classes: loads and stores. Finally, for miscellaneous instructions, there is only one class. In the *himode* scheme, a hybrid approach is used, which effectively collapses the thirteen classes of imode into seven classes. Data processing instruction sub-modes are now collapsed into two classes, based on distinction of the I bit value. Branch instructions sub-modes are now also collapsed into two classes, based on distinction of their direction (forward vs. backward branches). Data transfer and miscellaneous instructions are treated the same as in *imode*.

**State Model: using symbol position.** In a state model, the symbols are arranged into symbol sets based on their positions in the symbol stream, called *state* [5]. State, like context, also captures a form of higher-order correlation of a symbol to its location in the symbol stream. Hence, by using state, better compression can be obtained.

In this paper, a simplified notion of state is used: the position of a symbol in an instruction. In particular, the first symbol of an instruction is delivered to the first symbol set, the second symbol to the second set, and so on. The number of symbols of an instruction must be known to the decoder. If field-oriented symbols are used, the first symbol, which always contains the instruction mode, is always put into the first set. If any other symbol contains non-adjacent fields, its most significant field location is counted as its position.

**Hybrid Model: combining instruction mode and symbol position.** The two forms of modeling — context and state — can now be combined into a more refined architecture-oriented model. In this case, a symbol is classified not only by the current instruction's mode, but also by the symbol position in the instruction. In practice, better compression can be achieved if both of these two features are exploited.

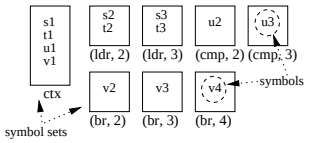


**Figure 7: A snapshot of symbol sets for Fig. 5.**

In this model, each symbol is labeled with both its instruction mode context and its position in the instruction. The label then defines a unique symbol set, corresponding to this hybrid model, and each symbol in the instruction stream is put into the corresponding symbol set dedicated to its label (i.e. hybrid model). The first symbol of an instruction, as in the context, is always put into its own dedicated symbol set. Each remaining symbol in the instruction is then placed in the appropriate symbol set, corresponding to the symbol's context (i.e. current instruction mode) and state (i.e. symbol position). The model can be used to separate heterogeneous frequency distributions hinged on both instruction modes and symbol positions, therefore capturing some key architecture-oriented correlations.

**Example.** The operation of hybrid model is further demonstrated by an example. Fig. 7 shows a snapshot of symbol sets after the symbols of the first four instructions in Fig. 5 are arranged by the model. In this example, instruction are grouped into thirteen classes by their modes, under the *imode* model, as mentioned above. Note that this classification is also used to decide symbol formations, which can be seen in Fig. 5. The four symbols, $s_1$, $t_1$, $u_1$, and $v_1$, which contain instruction mode information, are assigned to the symbol set labeled with *ctx*. For each instruction, in turn, the remaining symbols in the same instruction are deposited into various symbol sets for different context-position pairs. For example, the third symbol of `br`, v3, is put into the set labeled with `(br, 3)`. Due to space constraints, only some of the symbol sets used by the model are shown.

# 6. ENCODING

Two Huffman based encoding methods are proposed to construct codes for symbol sets. The first approach builds an *exact* Huffman code for each symbol set [11]. Each of the Huffman codes is exclusively tailored for a particular symbol set. The second new approach constructs a single heuristic *curve-fitting* Huffman code to be shared by all symbol sets so that the area overhead of the Huffman decoding circuitry is minimized. The code is derived in a way that, if only a single code is used for all symbol sets, it is the most efficient code.

**Exact Huffman Code.** In this method, for a symbol set, a Huffman code is derived based on the frequency distribution of symbols in the set. To encode a symbol, the Huffman code which corresponds to its symbol set is used. The symbol is encoded by the corresponding codeword in the Huffman code. It is easy to see that the encoding is optimal given the frequency distributions of each symbol set.

Although applying exact Huffman coding can produce optimal compression, it results in high decoding overhead: each Huffman code requires a hardware Huffman code analyzer. To reduce this overhead, a new approach, called *curve-fitting Huffman code,* is proposed.

**Curve-fitting Huffman Code.** A curve-fitting Huffman code is a single Huffman code constructed specifically to encode multiple symbol sets. Its derivation ensures that if only one code is used to encode multiple symbol sets, it is the optimal one. The benefit of using curve-fitting is lower hardware overhead: a single hardware Huffman analyzer suffices. Although compared to exact Huffman coding, a small amount of compression quality may be lost due to use of a single shared Huffman code, the code is designed optimally to minimize the impact on compression.

The coding efficiency of curve-fitting Huffman code is achieved by using a fundamental coding principle and exploiting the optimality of Huffman codes. The principle requires that more frequent symbols are encoded by shorter codewords. Furthermore, by this principle, the frequency distribution on which the single Huffman code is based can be synthesized. The synthesis of the combined frequencies ensures that the constructed Huffman code encodes all symbol sets optimally.

For encoding, the basic idea is to map each distinct symbol set to a single shared set of "pseudo symbols". The set of pseudo symbols is then mapped to a Huffman code. Hence, a single Huffman code, in

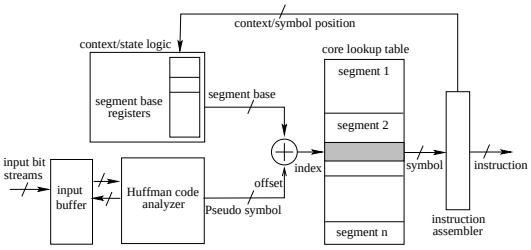


**Figure 8: Block diagram of the hardware decoder.**

this curve-fitting scheme, may correspond to many different symbols (at most one from each symbol set). For decoding, a given Huffman codeword is translated in the usual way to its corresponding symbol — in this case, a pseudo symbol. However, a second translation step is then required, to map the pseudo symbol to the original symbol which it represents. Since the pseudo symbol may map to many symbols, a record is continuously maintained of which lookup table to use (i.e. by updating an appropriate base register as context and state vary, see Section 7).

In more detail, the first step in encoding is to define a single new set of *pseudo symbols*. The symbols in each distinct symbol set will be mapped onto this set of pseudo symbols. These pseudo symbols are simply integers starting from 0. The number of pseudo symbols equals the cardinality of the largest symbol set. For example, if $M$ represents the size of the largest symbol set, then the highest pseudo symbol number is $M - 1$.

The second step is to map the symbols in each distinct symbol set onto these pseudo symbols. The mapping from each symbol set is straightforward, and is based on the frequency distribution of the symbols in that symbol set. In particular, the mapping assigns the most frequent symbol in a symbol set to pseudo symbol 0, the second most frequent symbol to pseudo symbol 1, and so on.

Finally, the third step is to encode the pseudo symbols using a Huffman code. To do so, the frequencies of the pseudo symbols must be derived. For pseudo symbol 0, its frequency is simply the sum of the frequencies all symbols which map to it (i.e., the sum of frequencies of the most frequent, or "first", symbol in each symbol set). Likewise, for pseudo symbol $n$, its frequency is the sum of the frequencies of all the symbols which map to it (i.e., the $(n + 1)$th-frequent symbols in each set). If the $(n + 1)$th symbol does not exist, it contributes a frequency of zero. This frequency distribution is then used to compute the optimal curve-fitting Huffman code.

# 7. DECODER DESIGN

A low overhead and efficient hardware decoder design is presented in this section. The design can be easily adapted to different symbol formation and modeling schemes. A single curve-fitting Huffman code is assumed during compression. Because only one Huffman decoding engine is needed for such an encoding, a simple hardware decoder can therefore be achieved.

The focus of this section is on the details of decoder itself. The cache and memory interfaces and address translations through the CLB/LAT lookups are not discussed. The input of the decoder is a sequence of compressed symbols, comprising a compressed cache block, fetched from the main memory. The output is a set of decompressed instructions, comprising the uncompressed cache block.

A high-level overview of the decoder components is outlined here. The block diagram is shown in Fig. 8. The input bit stream is stored in the input buffer. The bit stream is analyzed by the Huffman code analyzer. It takes in a maximum number of bits for a single codeword, and generates a pseudo symbol (as discussed in section 6). The pseudo symbol is used as an offset. It is added to the segment base supplied by the context/state logic unit; the base corresponds to the current context and state. The sum is the index of the real uncompressed symbol stored in the core lookup table. The generated symbol is then delivered to the instruction assembler, where the exact symbol composition is recovered and the instruction is assembled. The instruction modes and symbol positions are derived by the instruction assembler in the same way as performed by the compression tool. They are sent to the context/state logic to update the selected base register for the subsequent symbol, thereby providing its combined context and state (i.e. corresponding segment of the lookup table). The details of these components are further discussed below:

**Input buffer.** The Huffman codewords are read from memory, and are stored in this buffer for decoding. The buffer can accommodate the longest possible Huffman codeword. At all times, the first codeword in the buffer is aligned to the end of the buffer (see [6]). After a codeword is decoded by the Huffman code analyzer, its length is generated by the analyzer and sent back to the buffer. The length is used to retire the codeword from the buffer so the buffer can be updated to read the next codeword (see [6]).

**Huffman code analyzer.** The Huffman code analyzer takes in the contents of the input buffer and produces the pseudo symbol, which is a binary number. It also computes the actual codeword length. The length is used by the input buffer to retire the codeword. The design is based on the Huffman decoders proposed in [6, 7].

**Core lookup table.** The core lookup table is an implementation of the mappings between the pseudo symbol set and the sets of real symbols. The table is divided into a number of segments. Each segment is responsible for mapping symbols under a unique context and state, i.e. each segment handles one distinct symbol set. A segment has an associated base register, and maps the pseudo symbols to the corresponding symbols in the symbol set. The $n$th symbol stored in a segment is the $n$th frequent symbol in the corresponding symbol set. It is mapped to pseudo symbol $(n - 1)$. Since the symbols in a segment are listed in pseudo symbol order, therefore the symbols in the segment appear in decreasing frequency order.

The selection of a mapping, i.e. the changing of context and state, is accomplished by switching the base address of the segment storing the mapping. The base address is generated by the context/state logic, which outputs the appropriate base register value. The pseudo symbol is used as an offset to the base address. The sum of the base address and the offset is the index to the real symbol.

**Context/state logic.** The context/state logic effectively simulates the model used in compression, and generates the appropriate segment base address for each symbol in the instruction stream. The simulation is performed by a simple finite state machine, which is steered by the updates of instruction modes and/or symbol positions from the instruction assembler. When the current symbol is assigned to a certain symbol set in compression, the finite state machine selects the corresponding base address of the segment where the mapping from the pseudo symbols to the symbols in the set is stored. These base addresses of core lookup segments are stored in registers.

**Instruction assembler.** The instruction assembler is a buffer holding the decoded symbols and reconstructing instructions from the symbols. The assembling of instructions from symbols are performed by simple logic circuitry, which is designed according to how symbols are formed. The assembling logic can be easily derived in the design stage

The instruction mode and symbol position information are also derived by the assembler and sent to the context/state logic unit. The derivation is the same as performed in the compression. The derived instruction modes are also used to recover the exact symbol sizes and their formats. The context and state derivations can be implemented by simple logic functions.

# 8. RESULTS

In this section, the proposed code compression approach is applied in a number of experiments. Three different symbol formation schemes combined with various models of context and state are evaluated.

Our code compression scheme is implemented and applied to a set of benchmark programs. All but one benchmark, *espresso*, are from MediaBench [15]. The MediaBench is a collection of real-world programs for embedded systems. It is widely-used to evaluate code compression techniques, and for other embedded and media research problems. For fair comparisons, the toy examples in MediaBench are excluded, and only larger examples are included. All of the selected programs are larger than 10K bytes. For our experiments, the programs are compiled using gcc ARM compiler with optimization flags turned on. The system libraries are statically linked to programs, so each program is self-contained.

Fig. 9 shows the results of the experiments using various symbol formation schemes and model configurations. The chart shows overall ratios, consisting of compression ratios (cmp), ratios of core lookup

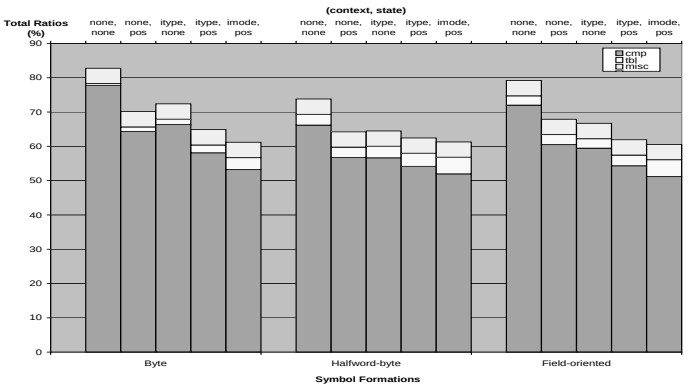


**Figure 9: Experimental results.**

| Benchmark | Orig. Pgm. Size (bytes) | context=imode, state=pos cmp/tbl/misc (bytes) | cmp/tbl/misc (%) | total (%) |
|---|---|---|---|---|
| cjpeg | 51700 | 25753/3102/2317 | 49.81/6.00/4.48 | 60.30 |
| djpeg | 69840 | 36234/3840/3123 | 51.88/5.50/4.47 | 61.85 |
| epic | 17940 | 8755/1719/799 | 48.80/9.58/4.46 | 62.84 |
| espresso | 112280 | 55559/3044/5047 | 49.48/2.71/4.50 | 56.69 |
| gs | 1791304 | 1003070/17569/80441 | 56.00/0.98/4.49 | 61.47 |
| gzip | 39520 | 19358/2200/1789 | 48.98/5.57/4.53 | 59.08 |
| libGL | 1420076 | 805121/10511/63789 | 56.70/0.74/4.49 | 61.93 |
| mpeg2dec | 39124 | 20028/2472/1759 | 51.19/6.32/4.50 | 62.01 |
| mpeg2enc | 58440 | 30646/3181/2620 | 52.44/5.44/4.48 | 62.37 |
| pegwit | 115500 | 59449/3989/5174 | 51.47/3.45/4.48 | 59.41 |
| pgp | 146692 | 75588/4051/6600 | 51.53/2.76/4.50 | 58.79 |
| rasta | 65312 | 34636/3479/2908 | 53.03/5.33/4.45 | 62.81 |
| toast | 27284 | 13131/1794/1217 | 48.13/6.58/4.46 | 59.16 |
| tsphere | 136724 | 70833/3786/6100 | 51.81/2.77/4.46 | 59.04 |
| unepic | 21212 | 10472/1793/961 | 49.37/8.45/4.53 | 62.36 |
| untoast | 27284 | 13131/1794/1217 | 48.13/6.58/4.46 | 59.16 |
| Avg. | 258764 | 142610/4270/11616 | 51.17/4.92/4.48 | 59.42 |

**Table 1: The results of the best combination of symbol and model configuration, using curve-fitting Huffman code.**

table sizes (tbl), and miscellaneous overheads (misc) including LAT and byte alignments, of each experiment run. Ratios are non-weighted arithmetic means across all benchmarks. In these experiments, three symbol formation schemes (byte, halfword-byte, and field-oriented), each combined with five model configurations (using different context and state), are evaluated, as labeled on the bottom (for symbols) and the top (for models) axis. All of the experiments use curve-fitting Huffman codes.

The five model configurations differ in context (i.e. level of granularity in classifying instruction modes) and state (i.e. whether symbol position is used). The **itype** context is the coarsest one, meaning that instructions are classified into four groups, each corresponding to a different instruction type. The **imode** is more fine-grained, indicating that instructions are classified into their individual sub-modes. Note that **none** indicates no context (or no state) is used. The bars labeling with **none, none** are schemes not using any modeling.

The exact symbol configurations selected for each of the three symbol formations require more elaboration. Among the experiments using byte (or halfword-byte) symbol formation, only one symbol configuration is used, as discussed in Section 4.

However in field-oriented experiments (as shown in the last group of bars in Fig. 9), two different symbol configurations are adopted. In the first four experiments, instructions are divided into symbols according to their instruction *types*, while in the last experiment instructions are divided into symbols according to their instruction *modes*. In particular, in the first four field-oriented experiments data transfer instructions, as well as miscellaneous instructions, are divided in the same way as the `ldr` instruction in Fig. 5; data processing instructions use the format of `cmp`; branch instructions use the format of `beq`. In the last field-oriented experiment based on instruction modes all six of the field partitions in Fig. 5 are used. Misc. and `ldr` instructions map to `ldr` format; `str` instructions map to the `str` format; `add`, `mov` and `cmp`, whether or not I bit is set, map to the corresponding field partitions in Fig. 5. Also, all four branch instructions map to the `beq` format regardless of their sub-modes.

The effectiveness of the experimented symbol and model configurations are now compared. In general, better total ratios are obtained by using more aggressive models regardless of symbol formations, as shown in Fig. 9. This indicates that the higher-order symbol correlations are truly captured by the proposed models. The field-oriented symbol generation scheme out-performs byte and halfword-byte hybrid symbol types only in hybrid models. This shows that the benefits of field-oriented symbols are only fully manifest in the more aggressive models.

Table 1 reports detailed results using the best combination of symbol and model configuration. The symbol configuration is field-oriented based on instruction modes, and the model is the hybrid model in which **imode** context and state is used. The table shows the compression results of each individual benchmark program, and the averages across all benchmarks. Column Benchmark reports benchmark program names; the next column (Orig. Pgm. Size) shows the original program sizes; then the break-downs of the results (cmp/tbl/misc) in terms of sizes and ratios are listed; finally the total ratios are presented in the last column. Among all configurations in the experiments, the best compression ratio achieved on a single benchmark program is $48.1\%$ (using the configurations in Table 1, see benchmark **untoast**), and the best overall ratio achieved (including decoder hardware overheads) is $56.7\%$ (see benchmark **espresso** in Table 1). The average overall ratio shown in Table 1 (59.4%) is also the best among all configurations.

The effectiveness of curve-fitting Huffman codes is also assessed. However, due to space constraints, the detailed statistics are not shown. Our experiments show that, on average, use of curve-fitting Huffman codes adds $2\%$-$3\%$ to compression ratios compared to use of exact Huffman codes. This indicates that the curve-fitting Huffman codes approximate the exact Huffman codes reasonably well, without incurring the additional hardware decoding overheads of multiple Huffman analyzers.

**Area overhead estimation.** We now give more details on the area overhead of the approach. First, the size of the core lookup table is included in our result table, since it is the dominating component in the decoder ("tbl"). Second, the LAT size is also reported, as well as the alignment overheads incurred in compressed memory storage, since each compressed memory block is byte aligned ("misc"). The overheads of other simple hardware components are not counted toward the results. Our estimation indicates their sizes are trivial, when compared with average uncompressed program sizes. For example, the CLB typically contains four to sixteen short entries (eleven bytes each) by design. Its overall size is fairly small. Likewise, the input buffer, context/state logic with base registers, the adder, and the instruction assembly buffer are implemented by small logic circuits. Their area overheads are thus ignored.

Finally, the only other non-trivial source of area overhead is the Huffman analyzer. The size of the Huffman analyzer depends on the size of the largest symbol set, or equivalently the number of pseudo-symbols generated. To estimate the size of the analyzer, we use the asynchronous Huffman decoder in Benes et al. [6] as a basic reference. Our estimation shows that in most cases, the size of our analyzer is two times the size of Benes' entire simpler Huffman decoder, due to the larger symbol sets we use when handling halfword and variable-length symbols (they use only byte symbols). Thus, the size of our analyzer is estimated as equivalent to 6K bytes, which is quite manageable. Following other published code compression approaches, this overhead is not counted toward the total ratio [27, 14, 23, 24, 19].

**Decoder latency estimation.** We now give a rough quantitative estimate of the latency of our decoder. The decoder latency is estimated as twice the decoder's in [6]. The critical component of our decoder, the Huffman code analyzer, has a similar architecture to Benes' decoder. However, the analyzer must handle roughly twice as many symbols. Furthermore, our decoder has one extra ROM on its critical path (the core lookup table). In sum, we estimate that our decoder latency should be manageable; further careful analysis is needed to provide a more accurate estimate.

**Estimated overall impact on system performance.** For the decoder's impact on overall system performance, based on Wolfe's study for a simpler but related decoder architecture [27, 14], we estimate only a small system-level performance degradation in the worst case using our approach. Wolfe's simulation for his simpler Huffman compression scheme yields two results: for a worst-case scenario (fast memory/high miss rate) up to 10% performance degradation; for a

best-case scenario (slower memory, also high miss rate), 2-3% performance *improvement*, since fewer bytes are transfered with compressed programs. We propose that Wolfe's results can be used as a basic estimate of our system-level performance overhead. Wolfe uses a different ISA (MIPS) and set of benchmarks. On the other hand, our compression ratios are significantly better, which suggests lower performance overhead. However, our decoder hardware itself is somewhat slower. Therefore, we conclude that our overall estimate is roughly in the same range as Wolfe's. and the degradation of the overall system performance is modest in the worst case.

# 9. RELATED WORK

In this section, a survey of related work on code compression is presented. The work is categorized into three groups based on their coding techniques. Dictionary-based approaches and statistical methods are introduced, then hybrid approaches, which combine both statistical and dictionary codings.

**Dictionary-based approaches.** There have been a number of recent code compression approaches using dictionary based techniques [21, 28, 16, 8, 9, 22, 25]. In dictionary coding, commonly-appearing symbol patterns are extracted from the symbol stream and each distinct pattern becomes a single entry in a dictionary. Each extracted pattern instance in the symbol stream is then replaced by a shorter pointer to its corresponding dictionary entry. Several of these approaches consider large symbols: using one or more instructions. Some of these statically identify common instruction sequences or "mini-subroutines", which are dynamically jumped to [21] or in-lined by the decoder [16]. The former approach does not preserve the original code but instead requires inserted jumps, while the later has complex decoding to store and inline variable-length multiple-instruction sequences. Others replace only individual instructions, either for the most dynamical frequent instructions [8], or a mix of dynamical and static frequent instructions [25], or all instructions [28]. In [8, 9] by Benini et al. the work primarily focused on low power on the DLX architecture; Netto et al. [25] further attempt to balance code size, power and performance for SPARC processors at the same time. The approach by Yoshida et al. [28] targeting ARM processors does not scale well with program sizes.

Several other dictionary based approaches replace a number of smaller symbols formed within an instruction. A limited form of architecture based symbols are used by Yoshida et al. [28], where an instruction is coarsely bi-partitioned to 'opcodes' and 'operands'. This approach provides a very limited notion of architectural fields. Only opcodes are replaced by dictionary indexes, and operands are left uncompressed. From their results, which did not directly indicate code compression, we derived roughly $80\%$ final average ratios on ARM programs, including dictionary sizes. Their work is improved by Ishiura and Yamaguchi [12] for a VLIW processor. Their algorithm optimally partitions instructions into variable-length symbols, where the sizes of a compressed program and its lookup dictionary are used as the cost function. In their work, a symbol can be of any form within an instruction, but does particularly target instruction modes and their fields. Others apply an adaptive, LZW based scheme to VLIW embedded systems. In the scheme by Lin, Xie, and Wolf [22], a sequence of byte symbols are replaced with an index to a dynamically constructed dictionary. Only limited amount of compression can be achieved in this way. Most of these approaches report only modest code size reductions. Their effectiveness depend heavily on application program characteristics and sizes.

In contrast, in this paper, we propose a fully architecture-oriented statistical approach to code compression. Our symbol formations consist of a discrete number of instruction fields, and varied field combinations and partitions are explored, unlike the above approaches. Although our symbol formations are less general than in [12], they are especially tailored for individual ARM instruction types and sub-modes, and can be combined seamlessly with powerful compression models to effectively capture higher-order correlations. Also, unlike several dictionary-based methods (e.g. [28]), our statistical approach uses a coding style whose effectiveness is not affected by program sizes. Our code compression scheme also aims for small decoding overheads. Besides, unlike most dictionary schemes, no modification is required to the compiler, and the uncompressed program is thus unmodified. The resulted decoder design is simple and easily extensible.

Another approach to code size reduction is designing a new instruction set with higher code density. Thumb instruction set defines new 16-bit fixed-size instructions for ARM processors [2]. By using Thumb $70\%$ compression ratio in average can be achieved. However, new compiler tools and modifications to the processors are needed for the new instruction set. In contrast, our approach requires none of the above changes.

**Statistical coding based approaches.** Others apply statistical coding techniques to compress the compiled programs [27, 14, 19, 20, 13]. Some use Huffman-based encoding [27, 14] as well as arithmetic encoding [19, 20]. In IBM's CodePack a less optimal coding, called "class coding", is adopted [13].

In the earlier statistical based approaches higher-order symbol correlations are not exploited. The scheme proposed by Wolfe and Chanin applies a static bounded Huffman code to byte symbols [27, 14]. In both papers, only modest compression results are reported. Kozuch and Wolfe [14] also suggest that variable-length symbols and higher-order entropies can be exploited. Two heuristics are proposed to select variable-length symbols, but no decompression hardware is proposed. Bar graphs in their paper suggest only a few percentage points improvement in overall compression. They also suggest that using higher-order modeling can further improve compressions, but this method is not explored due to its complexity, and only entropy evaluation was performed. Wolfe and Chanin [27] also first designed a practical decompress on cache refill decoder architecture. Based on Wolfe's framework, an asynchronous Huffman hardware decoder is proposed by Benes et al. [6], demonstrating efficient Huffman decoding implementations with small hardware overheads.

In later work, higher-order symbol dependencies are exploited to improve compressions. IBM's CodePack, which is a code compression solution for its PowerPC processors, applies a simple state model which uses symbol positions in an instruction as states. In particular, halfword symbols are used, and each instruction partitions symbols into two states: first halfword and second halfword. The first halfword and the second halfword of an instruction are encoded by two different codes respectively [13]. To reduce decoding overheads, only frequent symbols are encoded using variable-length "class" codes, which have two parts: prefix (which indicates length) and codeword. Later more complex Markov models, which uses not only symbol positions as states but also immediate previous symbols as context, are proposed by Lekatsas and Wolf [19, 20]. An unusual feature of their scheme is that bit symbols are used. The bit symbols are encoded by using reduced precision arithmetic coding, where a variable number of bits are encoded each time. The suggested context of a symbol is restricted to its four immediate previous symbols (i.e. four bits) for ARM processors (unlike our approach, which uses non-adjacent instruction fields as context). The number of states is recommended to be 32, the same as the instruction length. They report an average compression ratio of $56\%$ on ARM programs.

While introducing several novel ideas, the approach by Lekatsas and Wolf [19, 20] has several drawbacks. Though certain dependencies between bits are captured, correlations between farther symbols are missed due to the limited scope of their Markov model. Bit symbols also impose a constraint on the selection of the encoding method. These tiny symbols cannot be compressed by other codings like simple logarithmic codings. On the other hand, the arithmetic coding and bit symbols result in low decoding throughput. Their approach to dynamically locate compressed instructions requires branch instructions be modified so that they directly point to compressed targets in memory. This approach demands undesirable modifications to program data, branch instructions, and processor cores. Thus design time and cost are increased.

In contrast, the goal of this paper is to practically explore restricted forms of higher-order symbol dependencies. This is achieved by architecture based symbol formation, new models to capture higher-order symbol dependencies based on instruction modes, and a novel and cost-effective Huffman based encoding method. The modeling and encoding approaches are decoupled so different models and encodings

can be freely combined. In addition, to keep the processor core intact, the flexible decoder architecture proposed in [27, 14] is adopted. Our approach is targeted toward a specific architecture, though the framework can be easily adapted to other systems.

**Hybrid approaches.** Compression approaches combining dictionary and statistical codings are also proposed [4, 10, 3, 24, 23, 19, 18]. In most hybrid approaches, dictionary pointers are further compressed using statistical coding [19, 4, 10, 3, 23, 24, 19]. These approaches differ by what patterns of symbols can be put into the dictionary. The alternative is to compress instructions either by dictionary or statistical codings depending on the actual instruction types [18]. This latter approach is targeted to code compression of SPARC processors for low-power design. Lekatsas and Wolf [19] propose to recursively replace a short sequence of instruction opcodes or opcode-operand pairs, but their scheme requires more expensive Huffman decoding and iterative table lookups. They experiment with their scheme on MIPS and x86 instruction sets. Later expression tree based symbols are used to compress simpler MIPS and DSP architectures by Araujo et al. [4, 10, 3]. Based on this approach, Lin et al. [23, 24] propose to exploit limited operand dependencies across instruction boundaries to reduce dictionary sizes. Their approach also requires modifications to programs and micro-processors cores. They report in average $46\%$ compression ratios on ARM architectures [23]. The expression tree based symbols are used to capture *inter*-instruction dependencies. But they are quite large, spanning entire basic blocks. Because of their double compressions and large symbols, better results are reported.

While they tend to achieve better compressions, the hybrid approaches have several disadvantages in practice. These compression schemes require expensive hardware implementations. Multiple decoding engines for variable length codes, like Huffman codes, are needed [19, 23, 24, 4, 10, 3]. In some approaches, the lookup table sizes scale linearly with program sizes [4, 10, 3]. Dictionary lookup speeds are also slowed down due to large symbols. In the schemes where expression tree based symbols are used, original instructions need to be re-generated from decoded symbols. In contrast, our compression schemes are designed to allow simple, fast, and extensible hardware decoders. The compression efficiency is maintained by exploring *intra*-instruction correlations.

Finally, an alternative compression approach has been recently proposed by Lekatsas et al. [17]. This approach, called *Cypress*, combines both compression and encryption for multimedia systems in a unified framework, to enable design-space exploration. However, the focus of this work is not on new compression algorithms.

**Summary: comparison with leading approaches.** We now briefly compare our results to the leading results from the literature. The leading results targeting the MIPS processor report average compression ratios of $33.8\%$ [24], $52\%$ [19], and $53.6\%$ [3], respectively. Our average compression ratio for the ARM processor, including overheads, is $59.4\%$. It should be noted that the ARM instruction set is significantly denser than the MIPS instruction set, and has many more complex and varied instruction formats; the former is close to the code density of several leading CISC processors [26]. Hence, while a direct comparison is not yet possible, it is expected that these approaches will suffer significant penalties when applied to ARM. The best of these approaches [24] has much more complex decoder hardware than ours, and the processor core and the uncompressed programs must also be modified (unlike ours).

The leading compression results on the ARM instruction set are achieved by Lekatsas and Wolf ($56\%$) [20] and by Lin and Chung ($46\%$) [23]. Lekatsas and Wolf's results are only slightly better than ours, using more complex arithmetic encoding. They use statistical coding combined with a context and state based model to capture higher-order symbol correlations. While in our framework, versatile symbol formations, more aggressive models aiming for more powerful higher-order correlations, and various encoding schemes can be combined, this is not as easily feasible in their approach. Also, our proposed curve-fitting Huffman encoding allows faster and cheaper on-line decoding than using arithmetic encoding. Lin and Chung ($46\%$) [23] have the best reported compression results on ARM, using a hybrid model. This approach requires extensive hardware overheads, including a dictionary decoder and four distinct Huffman decoders (while our curve-fitting approach only requires one decoder). The overheads of these decoders appear to be significant (we estimate more than 10%), yet do not appear to be included in the final ratio. (Our analyzer is also not included, following reporting from other papers [27, 14, 23, 24, 19], but as indicated, is estimated at only 2-3% overhead.)

Finally, unlike our proposed approach, both of these two leading compression approaches require modifications to the processor core and to the uncompressed programs. Therefore, our compression results are the best that have been reported for ARM using a scheme which requires no modification to the compiler or processor core.

# 10. CONCLUSIONS AND FUTURE WORK

In this paper we present a powerful code compression scheme with a simple and modular decoder design. The compression scheme is architecture-oriented The proposed symbol formations, instruction mode and symbol position based models, combined with curve-fitting Huffman codes, allow us to capture higher-order symbol correlations with low decoding overheads. Further, the compression and decompression schemes are highly extensible.

The proposed scheme can be further explored in several directions. A more systematic and automatic way to select field-oriented symbols is worth pursuing. Other types of models targeting higher-order correlations inherent in a compiled program, like def-use chains and register usages, can be further investigated. In addition, different symbol encoding techniques with different compression and overhead tradeoffs, like arithmetic or logarithmic codings, can be evaluated.

# 11. REFERENCES


[1] Advanced RISC Machines Ltd. *ARM Architecture Reference Manual*, 1996.

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

[3] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression-tree-based algorithms for code compression on embedded RISC architectures. *IEEE Tran. VLSI*, 8(5):530–533, October, 2000.

[4] G. Araujo, P. Centoducatte, M. Cartes, and R. Pannain. Code compression based on operand factorization. In *MICRO-31*, pages 194–201, 1998.

[5] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, 1990.

[6] M. Beneš, S. M. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Async'98*, pages 43–56, 1998.

[7] M. Beneš, A. Wolfe, and S. M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *ARVLSI'97*, pages 219–236, 1997.

[8] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. In *ISLPED'99*, pages 206–211, 1999.

[9] L. Benini, A. Macii, and A. Nannarelli. Cached-code compression for energy minimization in embedded processors. In *ISLPED'01*, pages 322–327, 2001.

[10] P. Centoducatte, G. Araujo, and Pannain. Compressed code execution on DSP architectures. In *ISSS'99*, pages 56–61, 1999.

[11] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IEEE*, 40(3):1098–1101, September 1952.

[12] N. Ishiura and M. Yamaguchi. Instruction code compression for application specific VLIW processors based on automatic field partitioning. In *SASIMI'97*, 1997.

[13] T. M. Kemp, R. K. Montoye, D. J. Auerbach, J. D. Harper, and J. D. Palmer. A decompression core for PowerPC. *IBM J. Res. Dev.*, 42, 1998.

[14] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *ICCD'94*, pages 270–277, 1994.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30*, pages 330–335, 1997.

[16] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Micro-30*, pages 194–203, 1997.

[17] H. Lekatsas, J. Henkel, S. Chakradhar, and V. Jakkula. Cypress: Compression and encryption of data and code for embedded multimedia systems. *IEEE Design & Test of Computers*, pages 406–415, Sept.-Oct. 2004.

[18] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. In *DAC'00*, pages 294–299, 2000.

[19] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *DAC'98*, pages 516–521, 1998.

[20] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Tran. on CAD*, 18(12):1689–1701, 1999.

[21] S. Y. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *ARVLSI'95*, pages 272–285, 1995.

[22] C. H. Lin, Y. Xie, and W. Wolf. LZW-based code compression for VLIW embedded systems. In *DATE'04*, pages 76–81, 2004.

[23] K. Lin and C.-P. Chung. Code compression techniques using operand field remapping. In *IEE Proc.-Compute. Digit. Tech.*, pages 25–31, January, 2002.

[24] K. Lin, C.-P. Chung, and J. J.-J. Shann. Compressing MIPS code by multiple operand dependencies. *ACM Trans. on Embedded Computing Sys.*, 2(4):482–508, 2003.

[25] E. W. Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In *DAC'04*, pages 244–249, 2004.

[26] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, Thumb, and the ARM7TDMI. In *IEEE Micro Magazine*, pages 22–30, October 1995.

[27] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Micro-25*, pages 81–91, 1992.

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