Revised Version for Paper in ARVLSI 1995

Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques

Stan Y. Liao, MIT Dept. of EECS, Cambridge, Mass.

Srinivas Devadas, MIT Dept. of EECS, Cambridge, Mass.

Kurt Keutzer, Synopsys, Inc., Mountain View, Calif.

Abstract

We address the problem of code size minimization in VLSI systems with embedded DSP processors. Reducing code size reduces the production cost of embedded systems.

We use data compression methods to develop code size minimization strategies. We present a framework for code size minimization where the compressed data consists of a dictionary and a skeleton. The dictionary can be computed using popular text compression algorithms. We describe two methods to execute the compressed code that have varying performance characteristics and varying degrees of freedom in compressing the code. Experimental results obtained with a TMS320C25 code generator are presented.

1: Introduction

An increasingly common micro-architecture for embedded systems is to integrate a microprocessor or microcontroller, a ROM and an ASIC all on a single integrated circuit (Figure 1). Such a micro-architecture can currently be found in such diverse embedded systems as FAX modems, laser printers and cellular telephones. To justify such a level of integration these embedded system designs must be sold in very large volumes and, as a result, they are also very cost sensitive. The cost of the IC is most closely linked to the size of the IC and that is derived from the final circuit area. Surprisingly, it is not unusual for the bulk of the area of such ICs to be devoted to the ROM storing the program code for the microprocessor. In

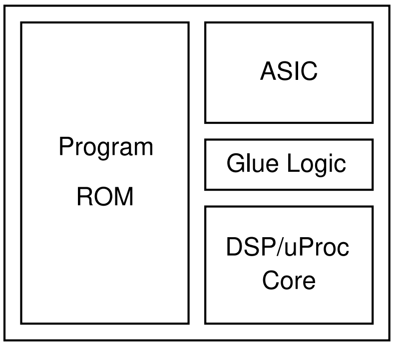

Figure 1. Heterogeneous System-on-a-Chip

these embedded systems the incremental value of using logic optimization to reduce the size of the ASIC is small because the ASIC circuitry is a relatively small fraction of the final circuit area. On the other hand, the potential for cost reduction through diminishing the size of the program is great. Alternatively, a given target die size for a product may limit the size of the ROMs and therefore the size of the code. In many embedded system projects, the ROM space estimated at the beginning becomes insufficient later in the development phase or during maintenance. Designers usually have to work diligently to reduce the code size in order to avoid excessive design modification [3, p. 18]. Consequently, diminishing code size may result in a significant productivity gain as well.

As the complexity of embedded systems grows, programming in assembly language and optimization by hand are no longer deemed practical or economical, except for time-critical portions of the program that absolutely require it. Recent statistics from Dataquest indicate that high-level languages (HLLs) such as C (and C++) are gradually replacing assembly language, because using HLLs greatly lowers the cost of development and maintenance of embedded systems. However, programming in a HLL can incur a penalty in code size. One reason for this is that compiler optimization techniques (for examples see [1]) have classically focussed on code speed and not code density. Also, most available compilers optimize primarily for speed of execution. Although some optimizing transforms such as common subexpression elimination can improve both speed and size at the same time, in many cases there is a speed-size trade-off. For example, subroutine calls take less space than in-line code, but are generally slower. Where execution speed is not critical (e.g., 80% of the code on which a typical program spends only a small fraction of time), minimizing the code size is usually profitable. For all of these reasons, we believe that optimizing for code compression is an important problem emerging from the integration of hardware ASICs and software (program memory) on ICs implementing embedded systems. Exploring new techniques to achieve code compression will be the focus of the remainder of the paper.

The paper is organized as follows. In Section 2 we will briefly review prior work on code compression and will introduce our own techniques. In Section 3 we will briefly review the model of data compression on which our framework is based, and state our conventions and assumptions. We then describe our approaches to code size minimization in Section 4, and describe the algorithm in Section 5. Experimental results obtained with a TMS320C25 code generator are presented in Section 6. We conclude in Section 7 with directions for future research.

2: Approaches to Code Compression

High-performance data compression techniques, such as [9], have been applied to reduce the size of executable programs. However, programs to be compressed are treated as sequential data, and before compressed programs can be executed, they are first decompressed and loaded into RAM. Thus, savings are achieved only on secondary storage (i.e., disks). These techniques are not directly applicable for code size minimization in the context of embedded systems since embedded systems typically do not have secondary storage. Furthermore, programs may have arbitrary control-flow structures. If we were to execute the program from its compressed form, we would need some mechanism of random access, which is impractical, if not impossible, for these compression techniques.

Recently, a novel RISC architecture which compresses instructions and then decompresses them in the cache was proposed in [8]. This approach, however, requires a significant redesign of the microprocessor architecture which is a stumbling block to its broad utilization.

The concept of data compression, however, inspired us to use a scheme that can achieve better results than mere conventional optimization. This scheme is based on a compression method which Storer and Szymanski [6] called an *external pointer macro* (EPM) model. In this model, compressed data consists of a dictionary and a skeleton. The dictionary contains substrings that occur frequently in the original data. The skeleton contains symbols from the alphabet of the original data, interspersed with pointers to the dictionary. This model is particularly suited for our approach to code minimization because the decoding process is simple and can be done in real time, and little or no extra hardware is required to support it. Thus, common sequences of instructions (not just common subexpressions, which may be elimininated by an optimizing compiler) are extracted and stored in a dictionary, and occurrences of these instructions are replaced by pointers (i.e., calls) to the appropriate location in the dictionary. An important characteristic of our approach is that the instruction set of the enhanced machine is a superset of the original machine; hence, all programs that could run on the original machine can also run on the enhanced machine.

In this paper we will present two methods of code size minimization. The first method is purely minimization on the part of the software; no hardware modification is necessary. In this method, common sequences are extracted to form dictionary entries and are replaced by calls to the dictionary. In addition, if a sequence is a *suffix* of some dictionary entry, it may also be replaced by a call to that entry with the appropriate starting point. We will generalize the suffix relation to a special class of blocks called extended blocks and show that this method applies under this generalization as well.

The second method is more flexible in its use of the dictionary. Unlike the first method, here the dictionary is seen as a large entry in itself, not a collection of entries. Any *substring* of the dictionary can be replaced by an appropriate pointer. Although some extra hardware is required to support this compression model, the total savings in code size are expected to outweigh the cost of the extra hardware.

As we shall see in Section 4, these two methods have different restrictions and therefore different strengths. Since the enhanced machine can execute programs written for the original machine, we can combine the two methods together to achieve greater reduction on code size.

We present a framework in which such strategies can be utilized for code size minimization. Any efficient text compression method that uses the EPM model will serve our purpose with a small degree of modification to take into account the control structures in programs, and our framework will benefit from any progress in such text compression techniques.

3: Preliminaries

3.1: Data Compression

We will briefly review the basic terminology of the macro model of data compression, as defined in [6]. The source data is treated as a finite string over some alphabet. In the *external pointer macro* (EPM) model, the compressed form of the source data consists of a *dictionary* and a *skeleton*. The dictionary is a string. The skeleton is a sequence of symbols of the alphabet interspersed with *pointers* to the dictionary. Each pointer represents a substring of the source data that is to be interpreted by the decoding process. A pointer consists of a pair of integers (a,l) where a indicates the position in the dictionary to which the pointer refers, and l indicates the length of the substring.

As an example, let the alphabet Σ be the set $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$ and consider the source string

$$x = \texttt{bbcdabbefabffbef}$$

with a dictionary

$$z = \texttt{abbef}$$

One compressed form of x using dictionary z is

$$y = \texttt{abbef . }(2,2)\texttt{cd}(1,5)(1,2)\texttt{ff}(3,3)$$

where "." serves as a delimiter for the dictionary. Assuming each pointer has a cost of 1 (and "." has cost 0), the ratio of the length of y to that of x is 13/16. The decoding process is straightforward: we simply scan through the skeleton, replacing each pointer by its reference to the dictionary with the indicated length.

Wagner [7] gave an algorithm based on dynamic programming for optimally parsing the source into a skeleton given a collection of phrases (similar to a dictionary, though less flexible), but did not show how the phrases were best generated. A heuristic algorithm for generating a dictionary was presented in Mayne and James [5]. Storer and Szymanski [6] showed that the problem of deciding whether the length of the shortest possible compressed form is less than k is NP-complete.

3.2: Definitions, Conventions, and Assumptions

We will model our system as a machine with a programmable processor, a program ROM, and some application-specific integrated circuit (ASIC). By *software* we mean the program stored in the program ROM, and by *hardware* we mean the processor and the ASIC. We will consider programs at the level of machine instructions, although the same techniques may be applied to intermediate representations and microinstructions. For intermediate representations this would entail an augmentation of the intermediate language to support the kinds of transformations described in the sequel.

We assume that subroutine linkage is accomplished with a link register or a stack of link registers. Thus the `CALL` instruction places the address of the next instruction in the link register and transfers control to the destination address, and the `RET` instruction transfers control back to the address designated in the link register.

Since the underlying compression model is one based on textual substitution, we must define what our alphabet is. To this end, we classify instructions into two types: *control-flow instructions* and *operational instructions*. Conditional branches, unconditional jumps, subroutine calls, and returns from subroutines, and instructions that modify the contents of the link register belong to the former type. All other instructions (i.e., load, store, and arithmetic and logical operations) belong to the latter. Our alphabet Σ consists of the equivalence classes of the set of operational instructions. Note that two instructions with the same operator but different operands are considered different instructions.

Unless otherwise stated, when we speak of a *graph* we mean a subgraph of the control-flow graph of the procedure in which each node is an instruction, and each edge denotes a possible flow of control between instructions. A graph is a quadruple (V, I, E, O) where V is the set of nodes, I is the set of internal edges, E is the set of edges entering G from the outside, and O is the set of edges leaving G. If $(u, v) \in E$, then u is called a *predecessor* of G, and v is called an *entry point* of G. If $(u, v) \in O$, then v is called a *successor* of G, and u is called an *exit point* of G. Two graphs G_1 and G_2 are *isomorphic* if (V_1, I_1, O_1) is isomorphic to (V_2, I_2, O_2), each pair of corresponding nodes denote the same instruction, and each pair of corresponding edges denote the same condition for control-transfer.

A *simple block* is a sequence of nodes v_1,v_2,...,v_k such that for $1 \leq i < k$, v_i is the only predecessor of v_{i+1}. A simple block that is not contained in any other simple block is called a *basic block*.

4: Proposed Compression Methods

The proposed methods are based on the external pointer macro compression model described in Section 3. The two methods differ in how the dictionary and pointers are represented, and each has its own strength and limitations.

4.1: Method I

The first method is an optimization purely in software. Common sequences are extracted and placed in a dictionary, and instances of these sequences are replaced by *mini-subroutine calls* to the dictionary. By mini-subroutine call we mean using a simple `CALL` instruction without passing parameters. Determining which sequences to extract is the core of compression algorithms. Unlike Method II, the sequences to be extracted are not restricted to basic blocks; some conditional branches may be allowed.

To characterize the circumstances under which conditional branches are allowed, we define the notion of extended blocks. An *extended block* is a graph G that has a unique successor. Note that under this definition single entry is not required; there may well be many edges coming into different nodes in the graph.

A *suffix* of an extended block G is a subgraph $G' = (V', I', E', O')$ of G such that if $u \in V'$, then all nodes $v \in V$ reachable from u are also in V'. The *suffix generated by a node* $u \in G$ consists of all nodes $v \in V$ such that v is reachable from u. Theorem 1 allow us to exploit the suffix relation to make better use of dictionary entries, as we shall see in the sequel.

Lemma 1 *If $G' = (V', I', E', O')$ is a suffix of an extended block $G = (V, I, E, O)$, then G' is also an extended block, and the successor of G is also the successor of G'. In other words, $O' = O$.*

Proof: Suppose G' is not an extended block. This means there exist edges (u_1, v_1) and (u_2, v_2) where $u_1, u_2 \in G'$ and $v_1, v_2 \notin G'$, $v_1 \neq v_2$. At least one of v_1 and v_2 must be in G, since otherwise (u_1, v_1) and (u_2, v_2) would be in O and G would not be an extended block.

Assume $v_2 \in G$. But since v_2 is reachable from a node in G' (i.e., u_2), by definition $v_2 \in G'$. This is a contradiction; hence, we conclude that G' is an extended block.

Furthermore, if the successor of G' were different from that of G, this would imply that G has two successors, which again leads to a contradiction. ■

Theorem 1 *Let H be an extended block whose successor is the* `RET` *instruction. Suppose $G = (V, I, E, O)$ is isomorphic to a suffix of H. We may, without altering the semantics of the program, replace G by $\hat{G} = (\hat{V}, \hat{I}, \hat{E}, \hat{O})$, which is derived as follows:*

- *For each entry node v of G, we have a node $\hat{v}$ in $\hat{V}$, which is a* `CALL` *instruction to v (the corresponding node of v in H).*
- *$\hat{I}$ =empty set.*
- *$\hat{E} = \{(u, \hat{v}) \mid (u, v) \in E\}$.*
- *$\hat{O} = \{(\hat{v}, s) \mid \hat{v} \in \hat{V}, s$ is the succesor of $G'\}$.*

In other words, H is the graph for a dictionary entry, and any extended block that is isomorphic to a suffix of H can be replaced with `CALL`*s to the appropriate entry points.*

Proof: Let v_1 be an entry point of G and s be the successor of G. Let $p = (v_1, ..., v_k)$ be a path (which may contain loops), where v_k is an exit node. Note that $v_i \in V$ for each i. Now let w_i denote the corresponding node of v_i in H. By isomorphism, each node w_i denotes the same instruction as v_i, and each edge (w_i, w_{i+1}) denotes the same condition for control transfer as (v_i, v_{i+1}). Hence, the sequence $q = (w_1, ..., w_k)$ is semantically equivalent to p.

Conversely, for any path $q = (w_1, ..., w_k)$ in H where w_1 corresponds to an entry point v_1 of G, there is a corresponding path $p = (v_1, ..., v_k)$ in G since G is isomorphic to a suffix of H (all nodes reachable from w_1 must be in the suffix, by definition).

Hence, by making a `CALL` to w_1 whenever we would enter G at v_1, and then returning to s, we preserve the semantics of the program. ■

Figure 2 shows some examples of extended blocks. In the figure, the circles denote basic blocks, and the squares denote (for the dictionary) entry points and (for the main program) dictionary calls to the corresponding entry points. In particular, $\boxed{r}$ denotes return from the mini-subroutine. Figure 2(d) shows a dictionary entry, and Figures 2(e)–(g) show the graphs resulting from replacing the extended blocks in (a)–(c) by mini-subroutine calls.

With an explicit `RET` instruction for each mini-subroutine, Method I allows for extended blocks, because an extended block has a unique exit point, and, when extracted as a mini-subroutine, this unique exit point corresponds to the `RET`. Furthermore, Lemma 1 allows multiple entry points and suffices to be exploited. For example, the edges entering node b can be changed to a mini-subroutine call with entry at $\boxed{b}$, because the lemma guarantees that the suffix generated by b (which consists of all nodes reachable from b) has exactly the same exit point. Similarly, the suffix in Figure 2(c) can be replaced by a call with entry at $\boxed{a}$.

Note that this transformation is not captured in structured high-level languages, because there is no construct to make a subroutine call to the middle of a function. Thus this method can achieve code size reduction not possible by simply writing function calls in the source code.

The present implementation is limited to basic blocks. Under this restriction, extended blocks are just strings and the suffix relation reduces simply to the one in the string-theoretic sense. Even so, we still obtain quite encouraging results (see Section 6).

4.2: Method II

The second method is somewhat more flexible in the use of the dictionary entries. It corresponds directly to a hardware implementation of the EPM model of compression. That is, as a pointer in the model consists of an address and a length, so in the instruction that calls the subroutine (say `CALD`, for *call-dictionary*) the number of instructions to be executed from the dictionary is specified as well as the address. Hence the return from the dictionary is implicit, without an actual `RET` instruction. Clearly, in typical architectures an instruction such as `CALD` does not exist; therefore, we will need to augment the instruction set.

One restriction on this method is that the boundaries of basic blocks have to be observed. This is because the paths of conditional branches may not have equal length, and since the point of return is implied by the length parameter, we cannot in general determine exactly when to return. Consequently, when we apply text compression algorithms, we must be careful that sequences to be extracted consist of operational instructions only.

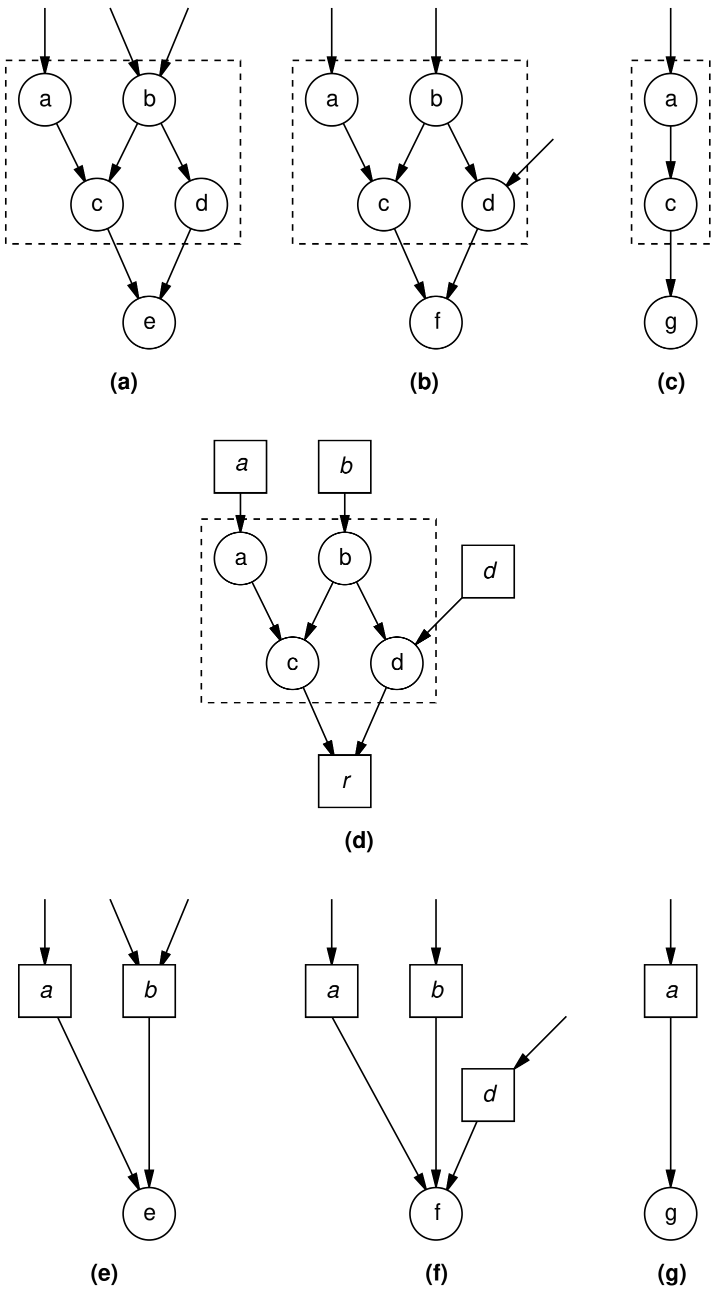

Figure 2. (a) (b) Isomorphic extended blocks. (c) An extended block isomorphic to a suffix thereof. (d) Extracting the extended block to form a mini-subroutine. (e) (f) (g) Extended blocks in the original graphs of (a), (b), and (c), respectively, replaced by mini-subroutine calls.

Extended blocks can be used if all paths from the entry to the exit have equal lengths. `NOP`s can be inserted so that the extended blocks will have this property. Although this may have an adverse effect on the dictionary size, it can potentially result in greater compression. The exact trade-offs are dependent on the application itself.

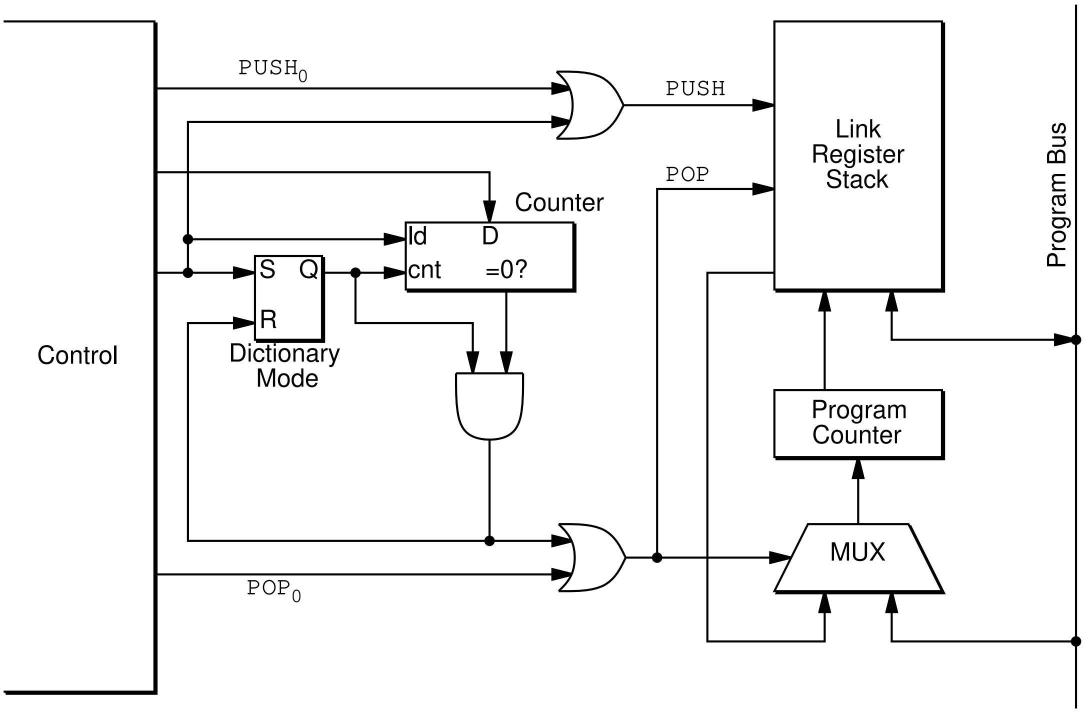

Figure 3. Architecture of Method II

We use the TMS320C25 architecture [4] to exemplify the method. The hardware modifications are shown in Figure 3 (for simplicity, much of the data path is not shown).

An S-R flip-flop, a counter, an AND gate, two OR gates, and a link register have been added to the base processor. The S-R flip-flop, if set, indicates that the processor is in dictionary mode. The counter records the number of instructions in the dictionary that remain to be executed. The link register stack is used for to store the return address for `CALD` as well as the original `CALL`. Hence, the `PUSH` signal for the link register file needs to be asserted for both `CALL` or `CALD`. Similarly, the `POP` signal for the link register file needs to be asserted when a `RET` instruction is encountered or when the counter reaches zero. The OR gates are used for this purpose.

With the hardware support, the steps of the `CALD` (*addr*, *len*) instruction are as follows:

1. Store the return address in the link register.
2. Set the processor in dictionary mode; load the counter with *len*.
3. Push the return address to the link register stack.
4. Set the PC to *addr*.

Once the processor is in dictionary mode, the counter begins counting down towards zero. When the counter reaches zero, the path from the link register stack to the PC is selected by the multiplexor, thereby accomplishing the implicit return. At the same time, the dictionary mode bit is reset to normal mode, and the processor continues in the main program. As an aside, we note that the implicit return as implemented works with pipelines as well. There is no complication with delayed branches because the PC is loaded with the return address directly.

Another advantage of the second method over the first is that the number of cycles required for dictionary accesses is reduced. Indeed, a dictionary access uses only one extra instruction: `CALD`. Provided that the hardware modification does not increase the critical path, this method incurs less performance penalty that the first. Although additional hardware means that off-the-shelf processors cannot be used, we believe that, as code size minimization becomes more important, processors can be easily designed in this way with hardware support for mini-subroutines. We are currently building hardware models to evaluate the impact of this modification on area and performance.

5: Algorithm for Code Compression

Since the problem of computing the optimal compression for the EPM model is NP-complete, we shall not attempt to present an optimal algorithm. The process of code compression consists of three phases: dictionary entry generation, substitution, and dictionary generation. During dictionary entry generation, substrings that occur many times in the instruction stream are discovered. Their occurrences in the instruction stream are then substituted by symbolic pointers. Finally, the dictionary entries are processed to produce the dictionary, and each symbolic pointer is replaced by the appropriate instruction or instructions (e.g., `CALL` for Method I and `CALD` for Method II).

5.1: Generation of Dictionary Entries

The instruction stream is first divided into basic blocks, using the algorithm described in [1, p. 529]. Then each block is compared with every other block, as well as itself, for common substrings. A threshold on the minimum length T (for example, 3) of substrings is prescribed, so that only potentially beneficial substrings are extracted.

A simple, naïve algorithm is used to find common substrings. The operation of the algorithm is illustrated in Figure 4. The two blocks are placed against each other with every possible region of overlap, beginning with the first T instructions of the first block and the last T instructions of the second. The matching substring or substrings in this overlapping region are identified and stored in a table. The second block is then shifted to the right by one instruction, and the process is repeated until the last T instructions of the first block are reached. If a block is compared against itself, we disregard the case when it is aligned exactly with itself, since the matching substring would be the entire block.

The worst case running time of this algorithm is $O(n^2)$, where n is the total number of instructions. This can be easily shown by the following analysis. It is clear that the process of comparing two basic blocks of lengths l_1 and l_2 takes at most $l_1 \cdot l_2$ steps. Now assume that there are m basic blocks, of lengths $l_1, l_2, ..., l_m$. The total number of steps $S(l_1, l_2, ..., l_m)$ is thus given by:

$$
\begin{aligned}
S(l_1, l_2, ..., l_m) &= \sum_{i=1}^{m} \sum_{j=1}^{i} l_i \cdot l_j \\
&= \frac{1}{2}(l_1 + l_2 + \cdots + l_m)^2 + \frac{1}{2}(l_1^2 + l_2^2 + \cdots + l_m^2) \\
&\leq n^2
\end{aligned}
$$

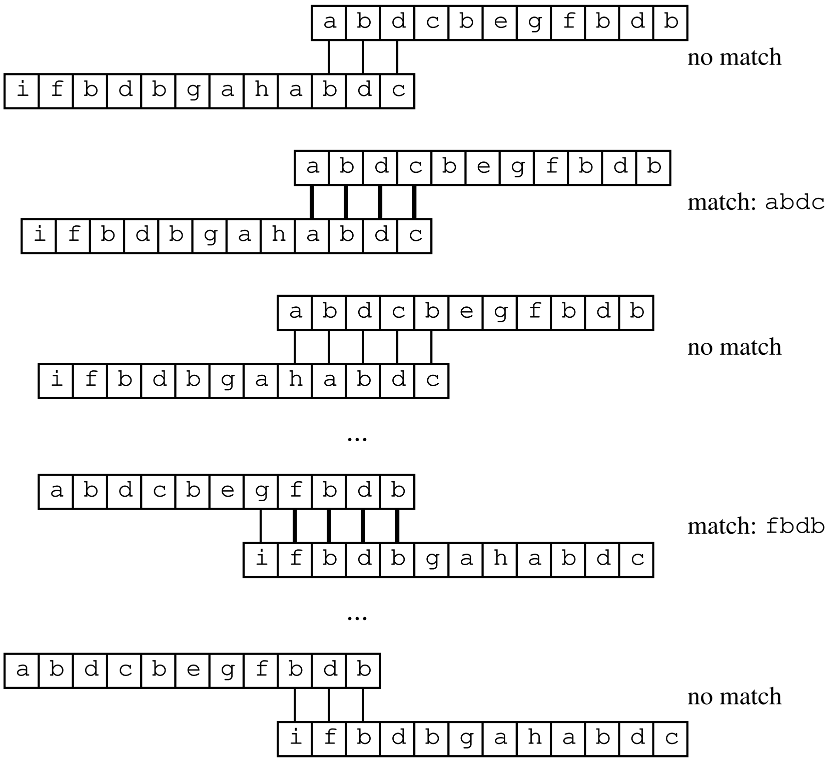

Figure 4. Identifying common substrings

5.2: Substitution

In the current implementation, a fast, greedy algorithm is used to replace occurrences of dictionary entries in the instruction stream by appropriate pointers to the dictionary. (The dynamic programming method suggested in [7] may yield improved results.) After dictionary entries are generated, we sort them according to their length. We then proceed to replace occurrences of each entry in the instruction stream by a symbolic pointer, beginning with the longest entry. At this time the actual values of the pointers are not available, because the dictionary is not yet put in its final form. Some entries may be eventually subsumed by some others and might not be used after all. Hence, the exact location and usage of the dictionary entries are not known until after the dictionary generation phase.

As we replace the occurrences of dictionary entries by symbolic pointers, we keep a count of the number of times each entry is used. This count is used, in the dictionary generation phase, to determine whether or not it is worthwhile to actually use this entry.

5.3: Generation of Dictionary

A dictionary entry i can be subsumed by another entry j if it is a suffix (Method I) or a substring (Method II) of entry j. In other words, we do not have to keep entry i in the dictionary because it is effectively available through entry j. In either case, entry i is said to be *subsumable* by entry j. Therefore, we do not have to keep all entries generated in Section 5.1, and can make the dictionary smaller. With this in mind, we will put together the dictionary entries to form the final dictionary as follows:

1. Beginning with the shortest entry, determine for each entry i whether it is subsumable by a longer entry j that is worthwhile by the criterion $S_j > 0$, where S_j is given in Eq. 1 (below). If so, we set L_i to 0 and mark entry i as subsumed.
2. Beginning with the longest entry, determine for each entry i whether or not it is worthwhile. The number of instructions we will save by using entry i is determined by the following formula:

$$S_i = n_i \cdot l_i - (L_i + n_i \cdot P) \quad (1)$$

   where n_i is the number of occurrences of the entry, l_i is the length of the entry (without counting the return instruction if it is present), L_i is the cost of the entry, and P is the cost of a pointer ($P = 1$ in this context). The first term in Eq. 1 denotes the total number of instructions of all occurrences of the entry before substitution, and the second term denotes the number of instructions after the substitution. Entry i is worthwhile if $S_i > 0$. Otherwise, we mark entry i as useless.

   Normally, for Method II, $L_i = l_i$; and for Method I, $L_i = l_i + 1$ (taking into account the return instruction). However, note that L_i may be set to 0 in step 1 above. Therefore, for example, whereas an entry of length three that occurs only once is usually not worthwhile, it could be if it is subsumable by another worthwhile entry.

   It may appear as though steps 1 and 2 are interdependent, since in step 1 we need to know if entry j is worthwhile, but in step 2 the usefulness of an entry depends on a value possibly changed in step 1. Actually, this presents no problem, because an entry can only be changed from being useless to being worthwhile, but not vice versa. Thus, if entry j is known to be worthwhile already from step 1, we can immediately infer that entry i may be subsumed by it. On the other hand, if entry j appears to be useless in step 1 but later is determined to be worthwhile, this means that there is another worthwhile entry k that subsumes entry j, and, by transitivity, entry i as well. Since there are a finite number of entries, either we eventually find a useful entry subsuming i, or no such entry exists. In the latter case, the entries that subsume entry i would also remain useless.
3. The dictionary is formed by concatenating the worthwhile entries. The subsumed entries and useless entries are disregarded, but the effective address (and length) of the subsumed entries in the dictionary are retained (for step 4). Of course, for Method II it is possible to exploit inter-entry substrings (substrings in the final dictionary that cross the boundary of two entries) and thus the order of concatenation may be of some consequence. Although the problem of finding an "optimal" order of concatenation is interesting in its own right, we conjecture that the amount of savings would not be significant.
4. In the instruction stream, symbolic pointers that point to worthwhile entries in are replaced by the appropriate instructions (`CALL` or `CALD` with the correct arguments). Those symbolic pointers that point to useless entries are substituted back by those entries, i.e., the original sequences are used rather than pointers.

6: Experimental Results

We present in this section some experimental results on example programs. We have obtained these results by applying our compression techniques on optimized code generated by TI's TMS320C25 compiler.

Example	uncompacted assembly code #instructions	compacted assembly code #instructions	dictionary size #instructions	reduction factor
RX	566	496	31	0.931
HILL	934	751	91	0.901
AIPINT	1423	964	222	0.833
COMPRESS	1994	1679	177	0.931
JPEG	2335	1775	331	0.893
GNUCRYPT	3731	2519	652	0.850
SET	4687	3458	582	0.862
CACHE	9499	7649	909	0.901
GZIP	10977	8555	1051	0.875
Cumulative	36146	27846	4046	0.882

Table 1. Results Using Code Compression Techniques (Method I)

RX and AIPINT are embedded state machine controller routines. SET is a collection of bit manipulation routines used in a DSP application. CACHE is a controller for a disk cache.

JPEG is an implementation of the JPEG image compression algorithm. COMPRESS and GZIP consist of core routines (i.e., without I/O) of the UNIX$^{\text{TM}}$ `compress(1)` and the GNU `gzip` programs, respectively.

Finally, HILL and GNUCRYPT are data encryption routines. The former is an encryption scheme based on matrix multiplication. The latter, from the GNU C Library, uses the data encryption standard (DES).

The statistics of the examples are summarized in Tables 1 and 2. In the table the numbers are given for instructions in the uncompacted assembly code, instructions in the compacted assembly code including dictionary pointers, and instructions in the dictionary. The size reduction factor is also given, which is the ratio of the total size of the skeleton and the dictionary to the size of the original code.

As the results indicate, reductions averaging 12% are obtained using Method I, and using the substring relation in Method II accounts for an additional reduction of approximately 4%. Depending on design goals and constraints, this 4% may justify the hardware modification required.

7: Summary and Future Research

We have presented a framework for the minimization of code size in VLSI systems containing embedded DSP processors. Our methods are based on data compression techniques. They offer different software/hardware design options and have different performance characteristics. The automatic minimization of code size relieves embedded system programmers of worrying about making programs sufficiently small, and thus allows them to enjoy the advantages of programming in high-level languages.

Several aspects of the current compression algorithm can be further improved. First, the current implementation observes boundaries of basic blocks. As we have noted in Section 4, Method I can accommodate extended blocks. Extended blocks pose a more difficult problem

Example	uncompacted assembly code #instructions	compacted assembly code #instructions	dictionary size #instructions	reduction factor
RX	566	488	32	0.919
HILL	934	725	77	0.859
AIPINT	1423	902	203	0.777
COMPRESS	1994	1615	180	0.900
JPEG	2335	1650	332	0.849
GNUCRYPT	3731	2433	558	0.802
SET	4687	3287	533	0.815
GZIP	9499	7295	967	0.863
CACHE	10977	8119	1051	0.835
Cumulative	36146	26514	3873	0.841

Table 2. Results Using Code Compression Techniques (Method II)

than basic blocks, because, while the set of basic blocks in a procedure is typically small, the number of extended blocks can be exponential with respect the number of instructions. Basic blocks are maximal with respect to well-defined boundaries, but a maximal extended block could be the entire procedure. We are currently developing efficient algorithms to identify potentially useful extended blocks.

At a higher level of compilation, the code generator can help out the code compression process by generating assembly code that has lower entropy. Chaitin [2] proved that the minimum size of a program is formally identical to the entropy of the sequence it generates. Thus, regardless of what compression model is used, programs of lower entropy can be made smaller.

For example, by permuting register or frame-relative offset assignment to variables, the code generator may be able to generate more isomorphic extended blocks. This is a combinatorial optimization problem, because while permutation can create new opportunities, at the same time it can cause other opportunities to disappear. We will formulate this problem more precisely and develop heuristics for solving it in the future. Information from the source code (such as the use of macros) and the intermediate forms will provide "hints" for arriving at good solutions. It is possible to apply our techniques also to intermediate forms; this will require the intermediate form to support the semantics of out-of-scope jumps.

Interaction with existing optimizions (for performance) also needs further study (the "phase-ordering" problem). For example, if heavy optimizations are performed before extraction of common sequences, opportunities for extracting may disappear because instances of the sequence in different contexts may be changed differently. On the other hand, if we extract common sequences first, there may be fewer opportunities for optimization. This is a topic we will continue to investigate.

Acknowledgments

The authors are grateful to the reviewers for their helpful suggestions. This research was supported in part by the Advanced Research Projects Agency under contract DABT-63-94-C-

0053 and in part by a NSF Young Investigator Award with matching funds from Mitsubishi Corporation.

References

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

[2] G. J. Chaitin. A Theory of Program Size Formally Identical to Information Theory. *J. ACM*, 22(3):329–340, July 1975.

[3] J. G. Ganssle. *The Art of Programming Embedded Systems*. San Diego, CA: Academic Press, Inc., 1992.

[4] Texas Instruments. *TMS320C2x User's Guide*. January 1993. Revision C.

[5] A. Mayne and E. B. James. Information compression by factoring common strings. *Computer Journal*, 18(2):157–160, 1975.

[6] J. A. Storer and T. G. Szymanski. Data Compression via Textual Substitution. *J. ACM*, 29(4):928–951, October 1982.

[7] R. A. Wagner. Common Phrases and Minimum-Space Text Storage. *Comm. of ACM*, 16(3):148–152, March 1973.

[8] A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of Micro-25*, December 1992.

[9] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. on Info. Theory*, IT-23(3):337–343, May 1977.