Compressed Code Execution on DSP Architectures

Paulo Centoducatte and Guido Araujo
IC–UNICAMP
Campinas, SP 13083-970, Brazil
(ducatte,guido)@dcc.unicamp.br

Ricardo Pannain
II–PUC Campinas
Campinas, SP 13020-904, Brazil
pannain@zeus.puccamp.br

Abstract

Decreasing the program size has become an important goal in the design of embedded systems target to mass production. This problem has led to a number of efforts aimed at designing processors with shorter instruction formats (e.g. ARM Thumb and MIPS16), or that can execute compressed code (e.g. IBM CodePack PowerPC). Much of this work has been directed towards RISC architectures though. This paper proposes a solution to the problem of executing compressed code on embedded DSPs. The experimental results reveal an average compression ratio of 75% for typical DSP programs running on the TMS320C25 processor. This number includes the size of the decompression engine. Decompression is performed by a state machine that translates codewords into instruction sequences during program execution. The decompression engine is synthesized using the AMS standard cell library and a 0.6μm 5V technology. Gate level simulation of the decompression engine reveals minimum operation frequencies of 150MHz.

1. Introduction

Embedded systems are computer systems designed to specific application domains. Because they are aimed at mass production, minimizing the final cost of such systems is a major design goal. Therefore, embedded system designs are typically constrained by stringent area, power, and performance budgets. In order to reduce the total system cost, designers are integrating memories, microprocessor cores and ASIC modules into a single chip, a methodology known as *System-On-a-Chip* (SOC). Microprocessor cores are selected based on the target application. In areas that require intense arithmetic processing, as in telecommunications, *Digital Signal Processors* (DSPs) have been the processor of choice.

A considerable part of the design effort of an embedded system is devoted to programming the application. Due to performance constraints, this task is predominantly done in assembly. Programming and debugging embedded code is a hard and time-consuming task. With the increase in the size of applications, assembly programming has become un-practical and error-prone. Compilers like SPAM [11], RECORD [8], CodeSyn [10] and CHESS [4] have achieved some success in generating quality code from high-level language programs. Unfortunately, compilers can reduce program size only to some extent. On the other hand, embedded programs are growing considerably large, to the point where the size of the program memory has become the largest share of the final die area (cost). A way to reduce program size is to compress its instructions, using a decompression engine to generate the original code during instruction fetch. This paper proposes a compression algorithm and a decompression engine targeted to DSPs. The experimental work reveals a 75% average compression ratio[1] for a set of typical embedded programs running on the TMS320C25 processor.

This paper is divided as follows. Section 2 describes related work in the area of code compression. Section 3 details our basic compression algorithm. The decompression engine is described in Section 4 and the experimental results are analyzed in Section 5. In Section 6 we conclude the work.

2. Previous Work

The problem of file compression has been extensively studied [3]. Almost all practical dictionary based compression tools of today are based on the work of Lempel and Ziv (LZ) and its variations [3]. Unfortunately, algorithms derived from LZ are not suitable for real-time code decompression. In LZ, codewords are decompressed sequentially using as dictionary the string formed by all the symbols already decompressed. This is a major drawback if the codeword encodes a forward branch instruction. In the rest of this section we describe only those compression techniques that are suitable to efficient real-time code decompression.

Wolfe and Channin [12] proposed the *Compressed Code RISC Processor* (CCRP). Programs are compressed one

[1]Compression Ratio = Size of the compressed code / Size of the uncompressed code.

cache-line at a time using Huffman codewords and byte-long symbols. During a cache miss, compressed cache-lines are fetched from main memory, uncompressed, and stored into the cache. Instructions in the cache and main memory have different addresses. The CCRP uses a main memory table, *Line Address Table* (LAT), to map (compressed) main memory addresses to (uncompressed) cache addresses. In order to reduce the number of accesses to the LAT, a *Cache Lookaside Buffer* (CLB) is provided to store the set of most recently used LAT entries. The advantage of the CCRP approach is that the latency of the decompression engine is amortized across many cache hits. On the other hand, the use of a cache makes it very difficult to estimate the execution time of the embedded program. This is particularly important for embedded systems running time critical applications. Moreover, embedded programs are usually stored into fast on-chip memories. The average compression ratio achieved by CCRP on a MIPS architecture is 73%. This compression ratio does not consider the size of the decompression engine.

Wolf and Lekatsas [6, 7] studied two different methods for code compression. The best compression ratio is achieved by their SADC method. In SADC, symbols are associated to instruction opcode and operand fields. During compression, instruction sequences are selected and a stream of bits is derived for each sequence of instruction fields. Each stream is then encoded using Huffman codewords. The average compression ratio achieved by this method on a MIPS architecture is 51%.

Lefurgy et al [5] describe a compression technique based on dictionary. Common sequences of instructions are assigned to a codeword. A dictionary in the decompression engine stores the sequence of instructions at the address given by the codeword. The decompression is performed by retrieving the sequence of instructions from the dictionary. Because instructions are compressed, the target address of jump and branch instructions must be recomputed. In order to deal with that, Lefurgy et al divide the target address bits into two parts. The first part stores the address of the word where the compressed target is located. The second part corresponds to the target offset inside the word. The average compression ratio using this technique for the PowerPC, ARM and i386 processors were 61%, 66% and 74%.

Araujo et al [2] proposed a code compression technique for the MIPS architecture that resembles the one previously studied by Wolf and Lekatsas [6]. They differ on how symbols are selected from the instruction stream. In [2] program expression trees are decomposed into sequences of opcode and operand patterns, a method called *operand factorization*. Patterns are then compressed separately using Huffman codewords, resulting in a 43% compression ratio. In this paper we also use expression trees as the basis of our algorithm, but unlike Araujo et al, we do not decompose them.

Liao et al [9] were the first to study the code compression problem for a DSP processor. Similarly to [5], compressed instructions are stored into a dictionary. Liao's idea is to substitute similar instruction sequences by sub-routine calls. Instructions are represented by boolean variables, and instruction sequences are encoded as minterms. Hence, the problem of compressing the program can be formulated as a set-covering problem. Instruction sequences are then converted into call instructions to sub-routines in the dictionary. A mechanism based on a stack is used to minimize the penalty of the sub-routine return instruction. The average compression ratio achieved by this technique in the TMS320C25 processor was 82%. To the best of our knowledge, apart from the work of Liao et al [9], no other research has addressed the problem of executing compressed DSP code.

```
LT      *+,AR6
MPY     *+,AR5
LAC     static_3
APAC
SACL    static_3
```

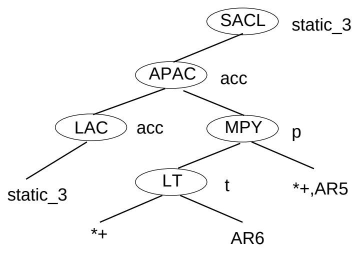

Figure 1. Expression trees.

3. The Compression Algorithm

Our compression algorithm encodes program expression trees using a simple binary code. First, instruction sequences are grouped into expression trees. An instruction is the root of an expression tree [1] if one of the following is true: (a) the instruction stores into memory; (b) the destination operand of the instruction is the source of more than one instruction inside the basic block; (c) the destination operand of the instruction is the source of at least one instruction outside the basic block; (d) the instruction is the first instruction in the basic block; (e) the instruction is a branch. We use expression trees as the basis for compression because compilers tend to generate similar expression trees during the translation of source statements, like *if-then-else* and *for*. Examples of two expression trees are shown in Figure 1.

We have selected a widely deployed DSP, the TMS320C25, as the target processor for this work. The TMS320C25 has 16 bit long instructions that are encoded using 15 different formats. Instruction formats use opcodes of different lengths (one or two words) and three addressing modes (direct, indirect and immediate). In the direct addressing mode the operand address is encoded into an instruction field. In the indirect addressing mode the operand

Program	Total Trees	Distinct Trees (%)
aipint2	928	295 (32)
bench	6693	2089 (31)
gnucrypt	1976	750 (38)
gzip	7171	2146 (30)
hill	627	292 (47)
jpeg	1124	572 (51)
rx	360	121 (34)
set	2798	969 (35)

Table 1. Number of distinct trees in a program. The numbers in parentheses are percentage with respect to the total number of expression trees.

Program Name (I)	# Opcode Seq. (II)	# Operand Seq. (III)	#Trees (IV)	Diff.(%) (V)
aipint2	62	291	295	1.4
bench	372	1997	2089	4.6
gnucrypt	193	718	750	4.5
gzip	403	2058	2146	4.3
hill	92	273	292	6.9
jpeg	146	560	572	2.1
rx	50	115	121	4.3
set	218	927	969	4.5
Average	192	867	904	4.2

Table 2. Number of distinct opcode (II) and operand sequences (III), when compared with the number of trees (IV).

format is `<ind,next>`, where `ind` is a side-effect operation with the current address register `AR`, e.g. auto-increment (`*+`), and `next` is the next current address register. In the immediate mode the value of the operand is encoded into the instruction field.

In order to measure the effectiveness of our algorithm we selected a set of example programs that are representative of the type of applications running on embedded processors and DSPs. Program *jpeg* is an implementation of the JPEG image compression algorithm, *bench* is a disk cache controller, *gzip* is a compression algorithm, and *set* is a collection of bit manipulation routines from a DSP application. Programs *hill*, *gnucrypt* are data encryption programs, and *rx* is an embedded state machine controlling routine.

We first generate *optimized* code for the example programs using TI's TMS320C25 compiler with optimization flag -O2. The resulting number of distinct expression trees for each program is shown in Table 1. On average, distinct expression trees correspond to 37% of all trees in the programs.

At this point, we want to determine what makes two expression trees different. Two expression trees are distinct if they have at least two different instructions. Two instructions are different if they have different opcodes and/or different operands. For example, instructions `ADD *+,AR3` and `ADD *+,AR3,0` have the same opcode but differ in one operand (0). In order to better understand what makes two trees distinct, we split each expression tree into the sequences of opcodes and operands that form it. For example, expression tree `[ADD *+,AR2,0 :  SUBK 16]` is decomposed into `(ADD,SUBK)` and `(*+,AR2,0,16)`. We then determine the set of distinct opcode and operand sequences for each program. Table 2 shows the number of distinct opcode (column II) and operand (column III) sequences for each program, as well as the number of distinct trees (column IV). Notice, from Table 2, that the number of distinct operand sequences (column III) is very close to the number of distinct trees (column IV). The difference of both (in percentage) is shown in column V. The average difference across all programs is approximately 4.23%. In other words, there is almost a one to one correspondence between an operand sequence and its expression tree. For the majority of the trees in a program, given an operand sequence there is only one opcode sequence associated to it. Therefore, the combination of its instruction operands is the main reason for the variety of trees in a program. In the table Table 2, we can note that the operand factorization method [2] has more redundanancy than our new method. This suggests that, at least for the case of DSPs, there is no need to encode operand and opcode sequences separately, as proposed in Araujo et al [2]. Trees should be encoded as an atomic unit.

The selection of the best algorithm to encode trees depends on their contribution to the program. In order to determine that, we ordered the set of distinct trees based on how frequent they show up in each program. The cumulative distribution of the distinct trees in the programs was computed. The result is shown in the graph of Figure 2. In the horizontal axis of the graph, trees are ordered in decreasing frequency. Notice from Figure 2 that the frequency distribution of distinct trees in DSP programs is very non-uniform. Actually, trees have exponential frequency distributions, as noticed in [5, 2]. In other words, a small set of distinct trees covers a large number of all trees in the program. In average, 70% of all program trees are covered by only 30% of the most frequent ones. This suggests that expression trees should be compressed using an encoding that assigns smaller (larger) codewords to (un)frequent trees. Figure 3 shows this encoding. In that figure the first bit of each codeword is an escape bit (b). When the escape bit is zero, the following k bits encode the 2^k most frequent expression trees. Quantity k is the minimum number of bits required to encode at least the 30% most frequent trees, i.e. $k = \lceil log_2 0.3N \rceil$, where N is the number of distinct expres-

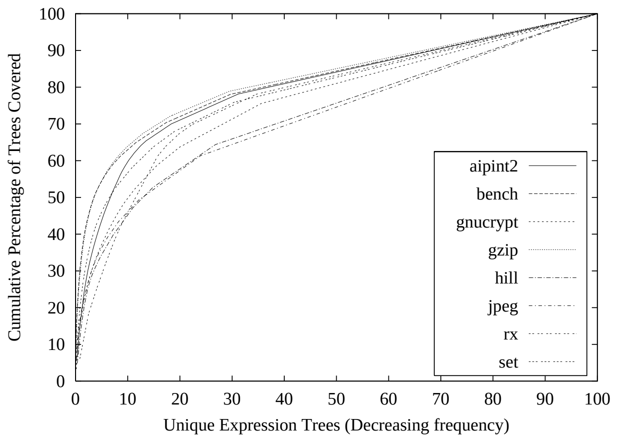

Figure 2. Percentage of program trees covered by distinct trees.

sion trees in the program. When the escape bit is one, the remaining bits encode a not so frequent tree. The number of bits required for that is $\lceil \log_2(N - 2^k) \rceil$.

The compression algorithm substitutes each expression tree in the program by its corresponding codeword. The resulting code is compacted so that all bits in every memory word are used. Although codeword compaction improves compression, it brings two other consequences. First, memory words can have more than one codeword. Second, codewords are allowed to split across memory word boundaries. In the first case, the decompression engine has to keep track of codeword boundaries inside the current memory word. In the second case, the engine must be able to put together pieces of a split codeword during two consecutive memory fetches.

b	CODEWORD

ESC BIT	CODEWORD LENGTH
0	$K = \lceil \log_2 0.3 * N \rceil$
1	$\lceil \log_2 (N - 2^K) \rceil$

Figure 3. Tree encoding

4. The Decompression Engine

Figure 4 shows the decompression engine for our compression algorithm. The majority of the work in the decompression engine is performed by the `Instruction Generator` (`IGEN`) state machine. During each machine cycle `IGEN` generates the bits that together form the various fields of an instruction from the expression tree being decoded. For the case of the TMS320C25 processor the output signals from `IGEN` are the instruction fields: opcode (`opcode`), immediate (`immed`), indirect (`ind`), next (`next`), compressed branch address (`caddr`), and uncompressed branch address (`uaddr`). The various fields of the instructions merge into the `Instruction Assembly Buffer` (`IAB`), where they are assembled and shipped to the CPU, at each machine cycle. Notice that one expression tree can have more than one instruction. Therefore, while the CPU executes the delivered instruction, a new instruction from the current expression tree can be decompressed without having to fetch another memory word. This is possible because module `IAB` contains a buffer to store the instructions that form the tree. The decompression cycle continues until all the instructions from the current tree are decompressed. In Figure 4 module `FETCH` is responsible for fetching the next compressed tree from main memory (see Section 4.1 for details).

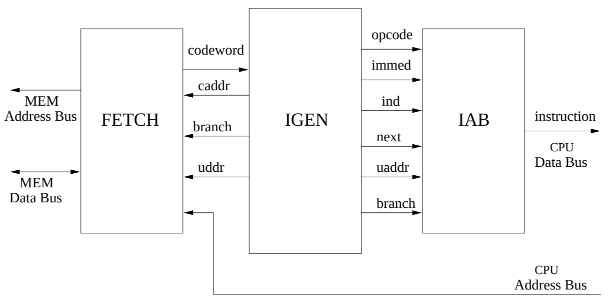

Figure 4. The Decompression Engine.

Module `IGEN` does not increase explosively with the program growth (see Section 5), as one might expect. There are two reasons for that. First, the number of state variables in `IGEN` is bounded by the size of the largest tree. When smaller trees are decompressed, the non-used state variables become don't cares. Moreover, many distinct trees have instructions with similar fields, which are eventually translated into shared logic in the `IGEN` machine. For example, distinct instructions `ADD *,AR3` and `ADD *-,AR3`, can share the same piece of logic in `IGEN`, given that both have to generate fields `opcode = ADD` and `next = AR3`. We have synthesized the state machine `IGEN` for each program in our benchmark, and the results are described in Section 5.

4.1. Codeword Fetch

Module `FETCH` fetches the memory word where the next codeword is (remember that one memory word can have more than one codeword), and extracts the codeword bits from it. It also aligns the remaining word bits

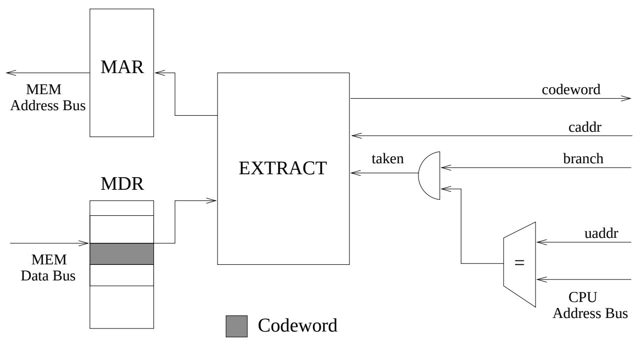

Figure 5. The codeword FETCH module.

so as to prepare the reading of the next codeword. This is done with the help of the `Memory Data Register` (`MDR`), `Memory Address Register` (`MAR`), and the `Extraction Logic` (`EXTRACT`) (Figure 5). `MDR` is used to store the memory word that contains the current compressed tree, while `MAR` stores the address of the next word in memory.

Because our compression algorithm allows codewords to split across the boundary of a memory word, the target of a branch instruction can start at any position inside a memory word. On the other hand, the CPU can only handle uncompressed addresses. In order to solve this conflict, `IGEN` generates two addresses, for each branch instruction that is decompressed. They are: uncompressed branch address (`uaddr`) and compressed branch address (`caddr`).

In the TMS320C25 architecture, a branch instruction has two 16-bit words, where the second word is the branch target. Whenever a branch instruction is detected, `IGEN` output signal `branch` is activated. It remains so, until the next instruction is decompressed. Module `IAB` appends the bits in `uaddr` to the other instruction fields generated by `IGEN`, so as to assemble the branch instruction (Figure 4). At the same time, the `EXTRACT` module uses the bits from `caddr` to determine: (a) the memory address of the word where the target compressed tree is; (b) the offset, with respect to the beginning of that word, where the tree starts. The address of the next expression tree depends on the result of the last branch instruction passed to the CPU. The `EXTRACT` module uses signal `taken` (Figure 5) to check that. Signal `taken = 1` if the last decompressed instruction passed to the CPU was a branch and the branch was taken, and `taken = 0` otherwise. Signal `taken` is determined by monitoring the progression of the CPU address bus. If during the next (program) memory read cycle the content of the CPU address is not equal to the last branch target passed to the CPU (i.e. `uaddr` $\neq$ `CPU Address Bus`), `taken = 0` and the next compressed tree is in `MDR`, or is the first tree of the next memory word. In the first case, the `EXTRACT` logic removes the tree from `MDR` and passes it to `IGEN` (Figure 4). In the second case, `MAR` is incremented, the next memory word is fetched and its first tree is extracted and passed to `IGEN`. If `uaddr = CPU Address Bus`, `taken = 1` and `caddr` is used by `EXTRACT` and `MAR` to fetch the next codeword. This approach allows the CPU to handle the same addresses as those in the original uncompressed program (`uaddr`), while the access to memory is performed using compressed addresses (`caddr`). By using this approach we do not require a modification of the processor address generation unit, like in [5].

5. Experimental Results

We tested our compression algorithm using a set of typical DSP programs. The compression ratio for each program was measured. We automatically generate VHDL code for the decompression engine, using the codeword assignment resulting from compression. The decompression engine was synthesized using the Leonardo Spectrum tools from Exemplar/Mentor Graphics, and the AMS standard cell library with a 0.6 μm, 5 V technology. For each decompression engine we target area optimization. We measured gate level estimates for the decompression engine area and maximum clock rate. The results are shown in Table 3. The average engine area was 3.3 mm^2.The average operation frequency was 167 MHz. These clock rates match those found in modern DSPs, suggesting that the latency of the decompression engine will not impact much the final performance of the system. From Table 3 it is also possible to determine how the size of the decompression engine depends on the program size. Although the area of the decompression engine increases as the size of the program grows (Table 3), the growth is sub-linear. This supports our observation that larger programs are much more redundant, and that similar expression trees tend to share the same logic gates in `IGEN`.

Program	Area (mm^2)	Clk. Rate (MHz)
rx	0.5	195
hill	0.8	165
aipint2	1.1	179
jpeg	2.1	150
gnucrypt	2.9	157
set	3.5	182
bench	7.5	155
gzip	8.1	161

Table 3. Area (mm^2) and maximum operation frequency (MHz) estimates for the decompression engine.

The compression ratio for each program is shown by the dark bars in the graph of Figure 6. The average program compression ratio was 28%. The final compression ratio has to take into consideration the size of the decompression

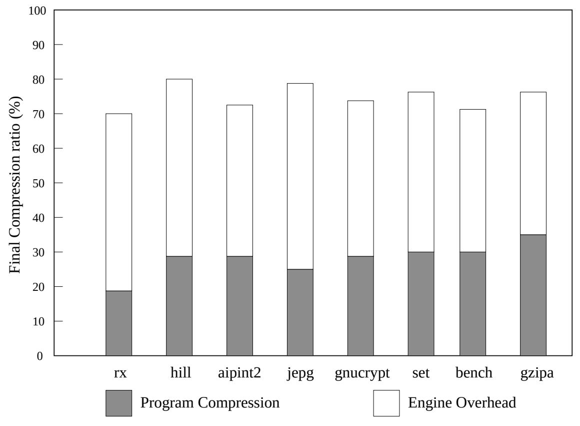

Figure 6. Final compression ratio.

engine though. We measured the area of the decompression engine and determined the engine overhead contribution to the final compression ratio[2]. The overhead of the decompression engine for each program is represented by the white bars in Figure 6. In average the decompression engine contributes 47% to the final compression ratio. The final compression ratio, once the engine size is taken into consideration was 75%.

Our final compression ratio (75%) is better than the best previous result on compression for DSPs, by Liao et al [9] (82%). On the other hand, the numbers do not differ much. In our approach we generate very dense code, using more area for the decompression engine. The opposite occurs in Liao's case. Their program compression ratio is higher, while the decompression engine is based on small dictionaries. The fact that we both obtain similar compression ratios, using radically different approaches, suggests that the minimum compression ratio that can be achieved for the TMS320C25 is around 70-80%. This is probably true for other DSPs as well. Similar experiments for RISC architectures [2] resulted in lower a compression ratio ($\approx 50\%$). We believe this difference can be explained by the fact that DSPs have highly encoded instruction sets, that are designed to maximize the usage of word bits. Given that instructions are already compacted, the efficiency of the (de)compression algorithm (engine) is jeopardized.

6 Conclusions

This paper proposes a code compression technique for DSP programs that improves previous work. Our approach is based on compressing expression trees. We show that trees have exponential distributions. We also propose a decompression engine that assembles instruction fields into uncompressed instructions sequences.

[2]i.e. Engine overhead = Decompression engine area / Area of the program memory for the original program.

7 Acknowledgments

The authors are grateful to Stan Liao for providing the test programs. We also thank the anonymous referees for their comments. This work would not be possible without the tools and resources provided by Mentor Graphics Corporation, through their Educational Program. This work was supported by a CNPq/NSF collaborative research grant and by CNPq Research Fellowship 300156/97-9.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1988.

[2] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression based on operand factorization. In *Proceedings of MICRO–31: The 31th Annual International Symposium on Microarchitecture*, December 1998.

[3] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Advanced Reference Series. Prentice Hall, New Jersey, 1990.

[4] D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[5] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proceedings of MICRO–30: The 30th Annual International Symposium on Microarchitecture*, pages 194–203, December 1997.

[6] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *Proc. of 35th ACM Design Automation Conference*, 1998.

[7] H. Lekatsas and W. Wolf. Random access decompression using arithmetic coding. In *Proc. of the Data Compression Conference*, March 1999.

[8] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, June 1997.

[9] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 1, pages 12–38, January 1999.

[10] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. CodeSyn: A Retargetable Code Synthesis System. In *Proceedings of the 7th International High-Level Synthesis Workshop*, Spring 1994.

[11] A. Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*. PhD thesis, Princeton University, May 1998.

[12] A. Wolfe and A. Channin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of MICRO–25: The 25th Annual International Symposium on Microarchitecture*, pages 81–91, December 1992.