Design of an One-cycle Decompression Hardware for Performance Increase in Embedded Systems

Haris Lekatsas
NEC USA
4 Independence Way
Princeton, New Jersey 08540
lekatsas@nec-lab.com

Jörg Henkel
NEC USA
4 Independence Way
Princeton, New Jersey 08540
henkel@nec-lab.com

Venkata Jakkula
NEC USA
4 Independence Way
Princeton, New Jersey 08540
jakkula@nec-lab.com

Abstract

Code compression is known as an effective technique to reduce instruction memory size on an embedded system. However, code compression can also be very effective in increasing processor-to-memory bandwidth and hence provide increased system performance. In this paper we describe our design and design methodology of the first running prototype of a one-cycle code decompression unit that decompresses compressed instructions on-the-fly. We describe in detail the architecture that enables decompression of multiple instructions in one cycle and we present the design methodologies and tools used. The stand-alone decompression unit does not require any modifications on the processor core. We observed up to 63% performance increase with 25% in average over a wide variety of applications running on the hardware prototype under various system configurations.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems-*Real-time and embedded systems*

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Designers in the high volume and cost-sensitive embedded system's market face many design challenges as the market demands single-chip-solutions to meet constraints like lower system cost, higher functionality, higher level of performance and lower power-consumption requirements for increasingly complex applications.

Code compression/decompression has emerged as an important field to address parts of these problems in embedded system designs. Typically, a system that runs compressed code is decompressing the code as it is needed by the CPU. The decompression can be performed outside the CPU (e.g. between L2 cache and main memory or between L2 cache and CPU) or inside the CPU as part of the instruction fetch unit or decoding unit. In any case only parts of the whole code will be decompressed at a time (on an instruction-level or block-level, for example) in order to minimize the memory requirements. The code is being compressed as a step following code compilation and then it is burned into Flash or ROM from where it is loaded during system reset into the main memory or L2 cache. That implies that the software cannot be changed and must be fixed.[1] If, however, the software application does change then the code has to be compressed again and the Flash or ROM has to be burned again. This is typically not a restriction for embedded systems that do not allow the user to change the software.

The primary goal in code compression[2] has traditionally been to reduce the instruction memory size although newer approaches have shown that it can even lead to performance increases of an embedded system.

Application areas for code compression/decompression are embedded systems in the areas of personal wireless communication (e.g. cell phones), personal computing (e.g. PDAs), personal entertainment (e.g. MPEGIII players) and other embedded systems where memory size and performance is a constraint like automotive control etc.

We introduce a code compression/decompression technology that is based on a commercial parameterizable processor core. We have designed a 1-cycle decompression unit that enhances the performance of the core system by an average of 25% and facilitating a code size reduction of 35% on average. Note, that the core has an already densely designed instruction set architecture as it's instruction size varies (2 instruction types exist: 16-bit instructions and 24-bit instructions). Our technology is generic as it can be applied to any RISC processor. The only application-specific parts are the processor and memory interfaces as far as the hardware is concerned as well as some modules of the software design flow. We have recently finished a hardware prototype that demonstrates the advantages of our technology in terms of performance increase and instruction memory size reduction.

This paper is structured as follows: Section 2 describes related work in code compression. Section 3 presents the compression and decompression algorithms and provides insight in the decompression process. Section 4 focuses on the hardware implementation and the design methodology we followed. Section 5 provides experimental results, while Section 6 concludes the paper.

[1]Note that this characteristic prohibits the system from using self-modifying code.

[2]Note that we use in the following the term "compression" or "decompression" when we denote a technology that runs code that has been compressed before the system runs and where code is being decompressed when the system runs i.e. on-the-fly.

2. RELATED WORK

In the following we report some of the most relevant work for code compression approaches.

Wolfe and Chanin developed the Compressed Code RISC Processor (CCRP), which was the first system to use cache misses to trigger decompression [9]. Their decompression engine is designed as part of the cache refill hardware. The instructions in each L1 cache block are separately Huffman-encoded so that each block can be individually decompressed without requiring decompression of other blocks in advance. The authors report a 73% compression ratio on the MIPS architecture.

CodePack is used in IBM's embedded PowerPC systems [3]. Their scheme resembles CCRP in that it is part of the memory system. The CPU is unaware of compression, and a table maps between the native and compressed address spaces. The decompression engine accepts L1-cache miss addresses, retrieves the corresponding compressed bytes from main memory, decompresses them, and returns native PowerPC instructions to the L1-cache. CodePack achieves 60% compression ratio on PowerPC.

Software decompression is also possible, simplifying the hardware design and allowing the decompression to be selected at runtime. Lefurgy et al. [6] proposed two hardware mechanisms to support software decompression. Another technique that can be carried out purely in software is a dictionary method proposed by Liao et al. [7] where mini-subroutines are introduced replacing frequently appearing code fragments.

Ishiura and Yamaguchi [4] proposed a compression scheme for VLIW processors based on automated field partitioning. They keep the size of the decompression tables small by producing codes for sub-fields of instructions. Benini et al. [2] limit the dictionary size by selectively compressing instructions. Okuma et al. [8] proposed an encoding technique that takes into account fields within instructions. Yoshida et al. [10] proposed a logarithmic-based compression scheme which can result in power reduction as well.

Our work differs from previous work in the following ways: 1) our decompression engine is capable of decompressing 1 to 2 instructions per cycle to meet the CPU's demand at no time penalty 2) it is completely transparent 3) it has been fully implemented and is running on a commercial FPGA board.

3. ALGORITHMS FOR THE ARCHITECTURAL DESIGN

In this section we describe techniques that enable the decompression engine to decompress in just one cycle. Though table-based dictionary encoding is a well known compression technique [1], advanced architectural design strategies are necessary to meet the one-cycle design constraint. Here is a summary of the most important constraints:

- Decompressing in one cycle
- Meeting the clock cycle delay budget
- No modification to the CPU should be necessary. That means the decompression engine has to interface to the CPU and to the memory hierarchy such that both CPU and memory hierarchy are not aware of the existence of the decompression engine.
- The decompression engine should be considerably smaller than the CPU.

Most of the architectural design strategies are independent of the CPU but some of them are especially designed for usage with our platform. A particular characteristc of the CPU we used is that the instruction set contains two types of instructions: 16-bit and 24-bit.

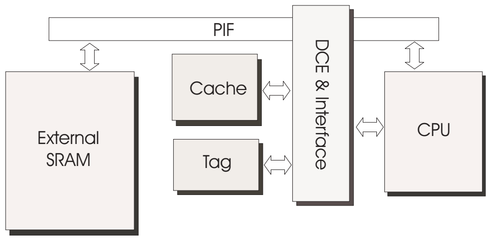

Figure 1: **Basic system architecture**

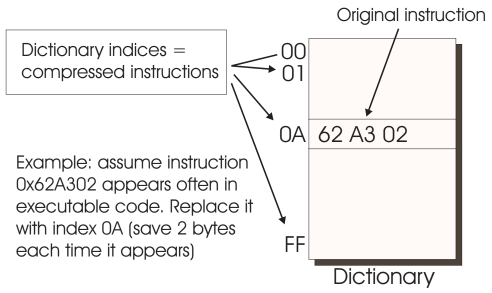
Original instruction
Dictionary indices = compressed instructions
00
01
0A
62 A3 02
FF
Example: assume instruction 0x62A302 appears often in executable code. Replace it with index 0A (save 2 bytes each time it appears)
Dictionary

Figure 2: **Principle of dictionary based compression technique**

3.1 Dictionary-based decompression table

We deploy a dictionary technique to compress frequently appearing instructions (according to program statistics). Dictionary coding refers to a class of compression methods that replace sequences of symbols with indices to a table. This table is called the "dictionary". The indices are the codewords in the compressed program. An entry in the dictionary holds either an original symbol or a sequence of symbols. This encoding reduces the data if the indices are shorter than the replaced symbols and if the dictionary is compact. The main advantage is that the indices are usually of fixed-length and thus simplify the decompression logic to access the dictionary and also reduce the decompression latency. Fig. 2 illustrates the dictionary compression algorithm. We use a dictionary table of 256 entries as a good compromise between compression ratio and table size. The table is duplicated in order to enable simultaneous decoding of two compressed instructions per cycle [3]. Due to the limited size of the table some instructions of the code are left uncompressed. We use two different encodings for 24-bit instructions, namely 8-bit and 16-bit codes. When we use 8-bit codes the whole codeword is used to index the decompression table. When we use 16-bit codes the 8 MSB bits are used to access the decompression table, while the 8 LSB bits are used to signal the type of instruction (i.e. whether it is compressed and what is its size). Our encoding allows us to use the 4 LSB bits of any instruction to determine its size.

The codes guarantee that differentiation between compressed and uncompressed instructions can be accomplished and decompression is done only when necessary. Fig. 2 shows the principle of dictionary encoding while the high-level system architecture is given in Fig. 1.

3.2 Decoding Tree Algorithm

The most significant part of the decompression algorithm is com-

[3]We found this to be more efficient than using a dual-port register file.

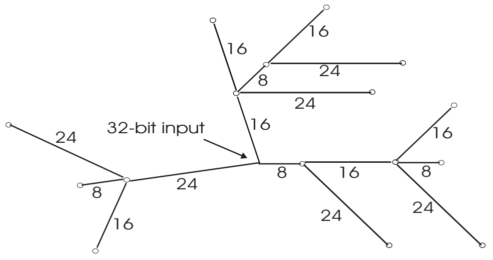

Figure 3: The structure of the 32 bit wide decompression tree

prised by the tree logic. The tree receives 32-bit instruction data from the cache (compressed) and produces decompressed instructions for the CPU. Since compressed data will expand to more than 32 bits, the tree also stores any extra decompressed or compressed bits for the next cycle.

The tree structure is shown in Fig. 3. It illustrates all the possible sequences of compressed instructions that can be subject to decompression by the decompression engine within any one cycle.

At each node we show all instruction sizes that can possibly occur. Since our technology compresses 24-bit instructions into 8-bit or 16-bit instructions, and since the commercial CPU we use has a mixture of 16-bit and 24-bit instructions, there are only three different sizes appearing in an actual compressed program. Hence, our tree starts with three branches corresponding to these sizes (8, 16 and 24). The root node is pointed to by the ”32-bit input” arrow. It then continues with all cases until 32 bits have been received (in certain cases some instruction combinations will not end on a 32-bit word boundary). After decompression any instructions/instruction-parts that do not fit in a 32-bit word boundary are buffered and sent to the CPU in the next cycle. The tree will reach terminal nodes when 32 bits have been collected to ensure that the CPU will always receive 32 bits per cycle. The basic registers and elements that we will use in the following to describe the tree logic are:

- We denote cw as a compressed 32-bit word coming from the cache, and $cw(i)$ is the i-th bit of cw.
- The register that holds data decompressed in the previous cycle is $prev\_decomp$. This is needed when, for example, the decompression engine decompresses two 16-bit instructions into their original 24-bit sizes and thus exceeds the 32 bit limit the CPU can handle. The remaining 16 bits of the second decompressed instruction are stored in the $prev\_decomp$ register for use in the next cycle.
- The register that holds data compressed in the previous cycle is $prev\_comp$. This is used when it is not possible to decompress the whole instruction in the current cycle because part of it belongs to the next word. It is therefore necessary to store the part of the compressed instruction that is available in the current cycle in the $prev\_comp$ register.
- Bytes to be sent directly from the next compressed data stream are held in register $bytes\_next$. Consider the case where there are two 24-bit uncompressed instructions. Since the decompression engine can only receive and produce 32 bits, it will only receive 8 bits of the second instruction from the cache. The bits needed to decode this instruction are always held in the first four bits. Therefore it is possible to know the type of the instruction but it is impossible to send more than 8 bits. The $bytes\_next$ register is used to inform the decompression engine that it needs to forward the next two bytes directly to the CPU in the next cycle without any decoding.
- We denote $output$ as the 32-bits that are sent to the CPU (this is not a register; the output pins are written to directly).

We differentiate between types of instructions by denoting their lengths as follows: 24 (a 24-bit uncompressed instruction), $24 \rightarrow 8$ (a 24-bit compressed to 8 bits instruction), $24 \rightarrow 16$ (a 24-bit compressed to 16 bits instruction) and $16(n)$ (a narrow instruction).

```
Tree algorithm
1)  If $(cw(3) == 0$ & $cw(27) == 0)$
2)    // 24 bits + 24 bits
3)    $output = cw(31-0)$
4)    $bytes\_next = 2$
5)    $prev\_decomp = empty$
6)    $prev\_comp = empty$
7)  Elsif $(cw(3) == 0$ & $cw(27) == 1$ & $cw(26-24) == 110)$
8)    // 24 bits + 8 bits compressed
9)    $output = Table(cw(31-24)) + cw(23-0)$
10)   $bytes\_next = 0$
11)   $prev\_decomp = 2\ bytes$
12)   $prev\_comp = empty$
13) Elsif $(cw(3) == 0$ & $cw(27) == 1$ & $cw(26-24) == 111)$
14)   // 24 bits + 16 bits compressed
15)   $output = Table(cw(31-24)) + cw(23-0)$
16)   $bytes\_next = 0$
17)   $prev\_decomp = 1\ byte$ (MSB bits of NOP)
18)   $prev\_comp = 1\ byte$ (from 16 bits comp. data)
19) Elsif $(cw(3) == 0$ & $cw(27) == 1$ & $cw(26-24) == xxx)$
20)   // 24 bits + 16 bits narrow
21)   $bytes\_next = 1$
22)   $prev\_decomp = empty$
23)   $prev\_comp = empty$
24) ...
```

Figure 4: Tree algorithm

We now present a fragment of the tree's algorithm structure. Listing all possible cases would take too much space (there are 25 different cases). In figure 4 we show four cases that should be enough for the reader to re-construct the whole logic. Consider an example with the following pattern of instructions coming from the cache: a 24, 24 → 8, 24 → 16, 16(n) (see Fig. 5). The above instructions would arrive at the decompression engine in two cycles of 32 bit streams.

1) Cycle 1: In the first cycle the decompression engine receives a 24-bit uncompressed and a 8-bit compressed instruction. 24 bits of the first instruction form the LSB 24 bits of the outgoing bit stream (to the CPU). The 8-bit compressed data is used to obtain the 24-bit instruction from the decompression table. The first byte of the resulting 24 bits of the decompression table is used to form the MSB 8 bits of the 32 bits outgoing bit stream to the CPU core, and the remaining 2 bytes are stored in the $prev\_decomp$ register. The dce_pc (i.e. decompression engine program counter; more details follow later) is incremented by 4.
2) Cycle 2: In the second cycle the decompression engine receives a 16-bit compressed instruction and a 16-bit uncompressed instruction. The decompression engine assembles the $prev\_decomp$ register's 16 bits as the LSB 16 bits of the outgoing 32 bit-stream. The LSB 16 bits of the incoming compressed instruction are used to obtain a 24-bit instruction from the decompression table. The resulting LSB 16 bits of the decompressed instruction's

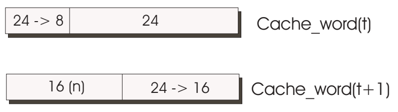

Figure 5: **Example for decoding using the tree**

24 bits are used to form the MSB 16 bits of the outgoing 32 bit-stream to the CPU core and the remaining MSB 1 byte is stored in the $prev\_decomp$ register. The MSB 16-bit regular narrow instruction from the cache memory is stored in the $prev\_comp$ register. The dce_pc is incremented by 4.

An important problem associated with code compression is how to locate branch/jump/call targets in compressed code since the original offsets will not point to "correct" locations after compression [4].

Our approach is to change branch offsets to point to the new compressed space addresses first proposed by Lefurgy et al. [5]. Branch offsets are patched after the compression phase. To simplify our design, we forced our targets to be aligned at word boundaries, as this simplifies the decoding tree. Patching the branch offsets involves a) compressing the original program and creating a map between native instruction addresses and compressed instruction addresses, and b) using the map to patch the offset-fields in the original branch instructions so they use offsets that correspond to the compressed address space.

An implication of using this technique is that the decompression engine must maintain its own program counter (dce_pc). When the application starts execution its value equals the CPU's program counter. As the execution progresses it diverges from the CPU's program counter. Using a separate program counter for the decompression engine also means that cache hits or misses must be triggered according to the dce_pc's value and not the $cpu\_pc's$ value. Our interface described in the next section takes care of this problem.

4. DESIGN ISSUES

Fig. 7 shows a block diagram of the decompression engine. The engine consists of the decoding logic and a 24-bit wide by 256 entries SRAM (decompression table). The Decompression engine's decoding logic is triggered by the arrival of incoming compressed or uncompressed 32-bit instruction words from the cache. The engine examines the first four bits (LSB bits), which form the select lines for the multiplexer in Fig. 7 and determine the type of the instruction.

The decompression table is accessed and the corresponding address contents return the decompressed 24-bit instruction. In some cases, there might be instances where the 32-bit compressed instruction stream may contain a part of an instruction and the remaining part might be in the next 32-bit compressed instruction stream. In that case the decoding logic provides two 16-bit registers ($prev\_decomp$ and $prev\_comp$) to store the previous portion of the instruction. These registers are shown as the pre_part_instr register in Fig. 7. Furthermore, two bits (pre_part_set signal in Fig. 7) are used to indicate that there is a portion of the previous instruction stored. In addition, a 2-bit register ($bytes\_next$) is used to indicate whether 8 or 16 bits of the pre_part_instr registers

[4]In the following we will use the term *branch* loosely to mean any instruction that alters the program counter such as a call, a jump, etc.

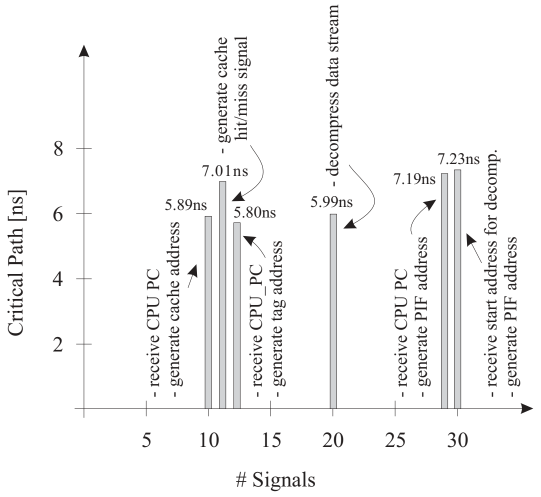

Figure 6: **Critical path**

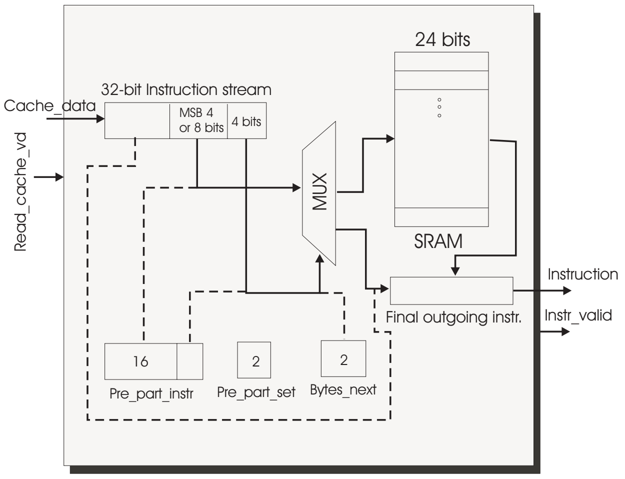

Figure 7: **Decompression block diagram**

comprise the previous instruction. The dotted lines in Fig. 7 denote alternative paths for MUX selection or for copying data from the input stream to the output register directly.

Fig. 8 shows the schematic of the decompression engine. The top part consists of the SRAM-based decoding table. The bottom part shows the circuits for the decompression engine program counter (dce_pc) the PIF (Processor InterFace) address generator and the cache/tag control logic.

Fig. 9 illustrates our design flow for building the decompression engine and for generation of the compressed executable. On the left side our proprietary tools (gray area) are shown which take a compiled object file and generate a compressed object file. This procedure involves one pass which gathers statistics, i.e. determines which are the most frequent instructions for inclusion in the dictionary. The second pass performs compression and also stores the most frequent instructions in the dictionary. The third pass handles the branch/call/jump targets. Linking and generation of an compressed executable code follows. Finally, it is loaded into the system SRAM.

Note that although a new decompression table must be loaded in

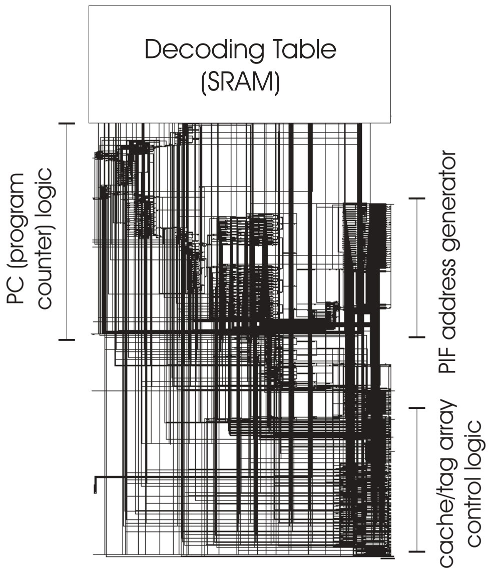

Figure 8: **Schematic of the decompression engine; the decoding table is SRAM based**

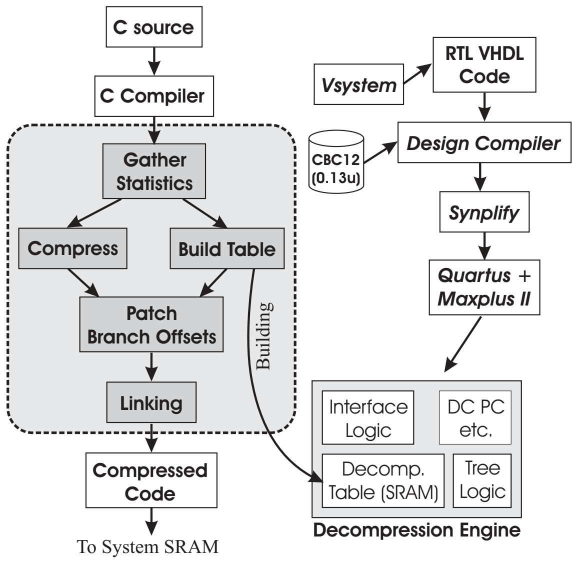

Figure 9: **Design Flow**

the decompression engine's SRAM every time a new application is compiled, there is no need to re-synthesize the decompression engine.

Here is a summary of the hardware design flow using commercial tools:

1) The decompression engine has been designed in VHDL and Verilog and simulated using Model-Tech's Vsystem simulator. For synthesis we used Synopsys' design compiler.
2) The CPU together with the decompression circuitry was synthesized using Synplicity's Synplify tool for Altera's APEX20KE family of devices.
3) The database format of Synplify was used by the Quartus and Maxplus II tools to place and route. The Quartus and Maxplus II tools generated the output format POF (Programmable logic Object File) and SOF (SRAM Object File) files of the design.
4) The generated POF files were then used to program Altera's APEX20KE device on the board.

5. RESULTS

The aim of the design of the code decompression unit was to guarantee that under no circumstances a delay of an additional clock cycle would occur since this would potentially prohibit its usage in a hard real-time system. Furthermore, the architecture should lead to a significantly increased system performance. Since we designed the decompression unit for a location between L2 cache and CPU both, instruction cache and main memory would profit from the compressed code. As a major effect, the instruction cache holds effectively more instructions and thus causing a higher cache hit ratio.

In order to increase the performance of the whole system we not only had to guarantee that the number of clock cycles is reduced compared to the non-compressed case, but we also had to avoid exceeding the cycle time budget. The results of this effort are shown in Fig. 6: it shows the six most critical paths in the design with the total maximum delay time for each path and the number of signals to be handled. It also gives the major functionality comprised by a certain path. An interesting observation is that the path *"decompress data stream"* (fourth bar) is not the most critical path as it accounts for only 5.99ns as opposed to 7.01ns for generating cache signals. We had a budget of 8.5ns for the whole clock cycle that would guarantee no additional delay. We managed to reduce this constraint by around 1.5ns (7.01ns to 8.5ns). The key in achieving this goal was to distribute the necessary computation equally among the most critical paths. The result can be seen in Fig. 6 where the six most critical paths are all within a time window of 1.12ns. Since the cycle time is guaranteed we can in the following concentrate to discuss the performance by solely comparing the number of clock cycles it takes to execute an application with and without code compression applied.

The applications have been chosen to demonstrate various effects and to show the advantages of our decompression architecture for diverse characteristics of applications like application area (video, animation, algorithmic), application domain (data dominated vs. control dominated) and size of the applications The applications are the following: the commonly available compress program (*"cpr"*) from SPEC95, a real–time Diesel engine control algorithm (*"diesel"*), an algorithm for computing 3D vectors for a motion picture (*"i3d"*), a real–time HDTV Chromakey algorithm (*"key"*), a complete MPEGII encoder (*"mpeg"*), a smoothing algorithm for digital images (*"smo"*) and a trick animation algorithm (*"trick")*.

Table 1 shows the performance results given as the number of clock cycles it takes to execute. Compared are the compressed cases to the un-compressed cases. Furthermore, each application's results are shown for various instruction cache sizes. The cache sizes have been chosen around the saturation point of the instruction cache i.e. the cache sizes below and slightly above the point where a further increase would not lead to a noticeable drop in cache misses. This point is typically the best choice for a designer as it represents a good compromise between costs (i.e. cache size) and obtained result (performance). As we can see from the table, the performance increases are significant as there is an average per-

Appl.	**Inst. Cache**	**Cmp**	**Exec Time [cycles]**	**Performance improvement [%]**
cpr	1k	no	187,777,618	n/a
	1k	yes	165,037,748	12.11
	2k	no	159,948,542	n/a
	2k	yes	155,885,849	02.54
	4k	no	154,460,263	n/a
	4k	yes	154,321,248	0.09
diesel	128	no	242,332	n/a
	128	yes	153,274	36.75
	512	no	233,441	n/a
	512	yes	85,136	63.53
	1k	no	84,724	n/a
	1k	yes	41,523	50.99
i3d	128	no	117,697	n/a
	128	yes	62,156	47.19
	512	no	58,475	n/a
	512	yes	42,418	27.46
	1k	no	44,158	n/a
	1k	yes	42,771	03.14
key	256	no	181,519,564	n/a
	256	yes	174,077,262	04.10
	512	no	178,888,128	n/a
	512	yes	174,755,812	02.31
	1k	no	174,318,687	n/a
	1k	yes	173,516,821	0.46
mpeg	2k	no	9,236,642	n/a
	2k	yes	5,024,733	45.60
	4k	no	5,142,418	n/a
	4k	yes	4,313,460	16.12
	8k	no	4,310,808	n/a
	8k	yes	4,271,149	0.92
smo	128	no	13,269,732	n/a
	128	yes	5,582,576	57.93
	512	no	3,658,805	n/a
	512	yes	3,657,890	0.25
	1k	no	3,716,882	n/a
	1k	yes	3,715,693	0.32
trick	256	no	5,416,167	n/a
	256	yes	2,347,908	56.65
	512	no	3,473,197	n/a
	512	yes	1,626,314	53.16
	1k	no	2,848,322	n/a
	1k	yes	1,551,766	45.52

Table 1: Performance results of all applications with instruction cache size as a parameter

formance boost of 25%.

A summary of the best results achieved for each application is given in Fig. 10 as a percentage improvement.

Note that the performance increase comes solely from the decompression hardware as any other system configurations in the comparisons are identical. In fact, in both cases (compressed and un-compressed) the hardware circuitry used is the same as the decompression hardware is simply bypassed in the non-compressed case.

6. CONCLUSIONS

In this paper we presented the design, architecture and prototyping hardware of a decompression unit that allows to run an embedded system using compressed code. It is the first running prototype that decompresses instructions in just one cycle and without any additional cycle time delay. The whole system is running as a hardware prototype comprising our decompression unit, a CPU, caches, memories and peripherals. Throughout a wide range of applications we observed a performance increase of up to 63% with 25% in average. Our technology is not limited to a certain CPU

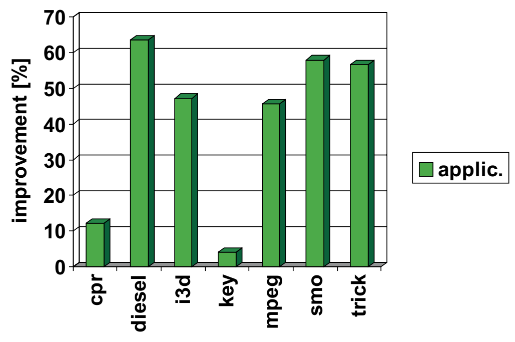

Figure 10: **Best performance improvements for each application**

and might achieve similar results when used with other CPUs. The limitation of our technology is that it can only be applied to systems that do not modify their instruction code during execution. This is the case for most embedded systems.

7. REFERENCES

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

[2] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. *IEEE/ACM Proc. of International Symposium on Low Power Electronics and Design (ISLPED'99)*, pages 206–211, 1999.

[3] IBM. CodePack PowerPC Code Compression Utility User's Manual. *Version 3.0*, 1998.

[4] N. Ishiura and M. Yamaguchi. Instruction Code Compression for Application Specific VLIW Processors Based on Automatic Field Partitioning. *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies*, pages 105–109, 1998.

[5] C. Lefurgy and T. Mudge. Code Compression for DSP. *CSE-TR-380-98, University of Michigan*, November 1998.

[6] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing Code Size with Run-time Decompression. *Proceedings of the International Symposium of High-Performance Computer Architecture*, January 2000.

[7] S.Y. Liao, S. Devadas, and K. Keutzer. Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. *Proceedings of the Chapel Hill Conference on Advanced Research in VLSI*, pages 393–399, 1995.

[8] T.Okuma, H.Tomiyama, A.Inoue, E.Fajar, and H.Yasuura. Instruction Encoding Techniques for Area Minimization of Instruction ROM. *International Symposium on System Synthesis*, pages 125–130, December 1998.

[9] A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. *Proceedings of the International Symposium on Microarchitecture*, pages 81–91, December 1992.

[10] Y. Yoshida, B.-Y. Song, H. Okuhata, and T. Onoye. An Object Code Compression Approach to Embedded Processors. *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, ACM:265–268, August 1997.