CoCo: A Hardware/Software Platform for Rapid Prototyping of Code Compression Technologies

Haris Lekatsas[1,2], Jörg Henkel[1], Srimat Chakradhar[1], Venkata Jakkula[1], Murugan Sankaradass[1]
[1]NEC Labs America, 4 Independence Way, Princeton, New Jersey 08540, USA
[2]Vorras Corporation, 1 West Drive, Suite 1202, Princeton, New Jersey 08540, USA
{lekatsas,henkel,chak}@nec-labs.com

Abstract

In recent years instruction code compression/decompression technologies have emerged as an efficient way to a) reduce the memory usage of an embedded system, b) to improve performance through effectively higher bandwidths and/or to c) reduce the overall power consumption of a system processing compressed code. We have presented efficient code compression/decompression techniques and architectures in the past. For the commercialization phase, we designed a novel hardware/software code compression/decompression platform (CoCo). It consists of a software platform that prepares, optimizes, compresses and compiles instruction code and a generic, parameterizable FPGA-based hardware architecture in form of a hardware platform that allows to rapidly evaluate prototypes of diverse compression/decompression technologies. We show the flexibility of CoCo, its ability to achieve code compression ratios (parameterizable) of up to 50% with a slight system performance gain and its ability to apply compression on real-world compiled code without any limitations where others have made implicit software-restrictive assumptions.

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

Keywords

Embedded Systems, Code Compression

1. INTRODUCTION

In recent years, diverse novel code compression/decompression technologies for embedded systems have been proposed. This trend is mostly driven by the advent of MPSOCs (Multi-Processor-System-On-Chip) designs. Not only do those SOCs often exceed a complexity of 100 million transistors on a single piece of silicon, but they also comprise a large set of heterogeneous or homogeneous multiprocessors (we will simply use the term *"Processing Unit"* in the following) that communicate via an on-chip network. Typically, those MPSOC designs feature a large shared memory that is accessed for diverse purposes by at least a subset of the multiprocessors. In addition, a single processing unit tends to have its own (medium size) memory that holds the code and data that is being accessed exclusively by this processing unit. Those individual memories can still have a size of several hundred KB to several MB each and add up to a size that exceeds by far the single shared memory.

An example for a homogeneous MPSOC is a state-of-the-art 64-channel TDMA chip that comprises eight CPUs each processing voice encoding on eight channels at a time. Each of the CPUs has its own instruction memory of 32KB. Even though the code in each of the instruction memories is almost identical, the simultaneous access pattern constraints to provide an individual memory for every CPU. Furthermore, the chip itself is constrained to $13mm^2$. This, together with the TDMA performance constraints would result in a mismatch, given a technology of 0.13μ.

Our solution to meet all constraints was to deploy a code compression technology: code in the instruction memory could be compressed and thus require a smaller instruction memory. If, for example, we compress the code by 50% this would save us $8 \times 16KB$ and we would meet the area constraints. We also found that code compression can even boost the performance of an MPSOC due to an effectively increased bandwidth and an improved memory access pattern in a multi-level memory hierarchy (e.g. a cache might appear larger since it holds compressed code and thus the cache hit ratio might increase).

In this paper we present a parameterizable hardware/software platform for code compression/decompression technologies that is the first of its kind. The software platform prepares, optimizes, compresses and compiles instruction code whereas the hardware platform provides a generic, parameterizable architecture to implement various compression technologies and to adapt them to diverse instruction set architectures. Among others, it allows us to tradeoff the compression ratio against the performance gain and thus it is a vital tool- suite to explore the benefits of code compression/decompression technologies under varying constraints and diverse MPSOC architectures.

The structure of this paper: Sec. 2 summarizes the most prominent work in code compression whereas Sec. 3 revisits issues in code compression as they are necessary for the understanding of the rest of the paper. It also states our contributions and assumptions. Our CoCo hardware/software platform is introduced in Sec. 4 along with two exemplary compression/decompression schemes that show the flexibility of CoCo. Experimental results obtained with CoCo are shown and discussed in Sec. 5, while Sec. 6 gives a summary with conclusion.

2. RELATED WORK

A summary of the most prominent work in code compression follows; we present work that is closer to our CoCo platform. Wolfe and Chanin proposed CCRP *Compressed Code RISC Processor*, which was the first system to use cache-misses to trigger decompression [9]. CodePack is a similar system used in IBM's embedded PowerPC systems [2]. Their scheme resembles CCRP in that it is part of the memory system. Software decompression is also possible, simplifying the hardware design and allowing decompression to be selected at run-time. Lefurgy et al. [5] 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. [6] where mini-subroutines are introduced replacing frequently appearing code fragments. Ishiura and Yamaguchi [3] 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. Xie [10] proposed another approach based on approximate arithmetic coding targeting VLIW code. Other methods based on dictionary coding include the work by Benini et al. [1] which selectively compresses instructions and the work by Lefurgy et al. [4] for DSP. Okuma et al. [7] proposed an encoding technique that takes into account fields within instructions. Yoshida et al. [11] introduced a logarithmic-based compression scheme which can result in power reduction as well.

All these approaches are targeted to a specific instruction set architecture and/or they make implicit assumptions that restrict the instruction code. Our CoCo platform overcomes these limitations as it is designed to work with diverse instruction set architectures and to evaluate various compression schemes rapidly through its flexible and parameterizable concept. In addition, it does not imply restrictions on the code.

3. CONTRIBUTION OF OUR WORK AND BASICS IN CODE COMPRESSION

In the following sub-section 3.1 we shortly revisit basics in code compression as they are mandatory for the understanding of the rest of the paper. For deeper information on these topics, please refer to [8, 9].

3.1 Revisiting Basics in Code Compression

- **Random Access**
  Code compression mandates *random access* i.e. the ability to start decompressing the code at any location that might be a target of a jump/call/branch instruction. Only if small parts of the code are decompressed at a time, the memory usage of a system can be kept low. In contrary, file compression/decompression (e.g. images) do not need random access.
- **Granularity in code compression**
  Due to the random access constraint, typical granularities i.e. code portions that are compressed independently, are *basic blocks* (i.e. the instructions in a basic blocks are always entirely executed in a sequential way from the first to the last instruction) or single instructions.
- **Indexing**
  Assumed that the code is decompressed before it is fed into the CPU[1], there are two address spaces: the compressed address space and the uncompressed address space. The Program Counter PC of the CPU will then operate in the uncompressed address space whereas the main memory etc. will operate in the compressed address space requiring a CPC (Compressed Program Counter). Indexing is the problem of mapping the PC to the CPC since code is not executed sequentially due to branch/jump/call instructions etc. Wolfe and Chanin [9] proposed using a mapping table. The overhead for such a table can be large when the granularity (see above) decreases. An alternative approach is to leave branch/jump/call instructions untouched during compression and to patch the offsets to point to compressed space after compression is done.
- **Dictionary/table-based compression algorithms**
  A category of data compression algorithms that use tables is dictionary coding. In dictionary coding, frequently appearing sequences of symbols are replaced by indices to dictionaries that hold the original symbols. Those indices are smaller in size than the original sequences, thus achieving a compression. One of the first dictionary techniques proposed was LZ coding by Lempel and Ziv [8].
- **Compression ratio**
  We define the *compression ratio* as the ratio of compressed code size to uncompressed code size. Example: a compression ratio of 70% would mean that the code size has shrunk by 30%. This ratio excludes indexing and dictionary tables and only shows raw code size reduction.

3.2 Assumptions and Basic Hardware Architecture of CoCo

The basic architecture of CoCo is shown in Fig. 1: the whole system except for the CPU is processing in the compressed space. This is beneficial since as many as possible system parts profit from the compressed code (caches and main memory can be smaller, buses have an effectively higher throughput etc.). The decompression engine is located as close as possible to the CPU. This is a so-called *"post-cache"* architecture as opposed to a *"pre-cache"* architecture where the instruction cache would not profit from the compressed

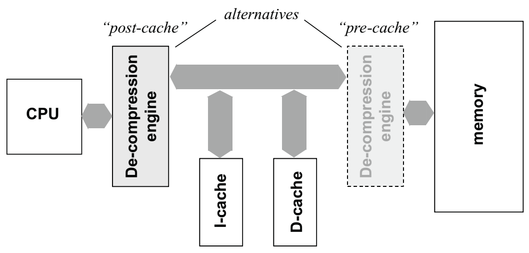


Figure 1: Example for an architecture the COCO platform is aimed at

code (shown in dashed lines in Fig. 1). In addition, it can be seen that the decompression engine is separated from the CPU meaning that no modifications to the CPU are necessary.
A further characteristic of CoCo is that decompression is performed on-the-fly i.e. whenever an instruction(s) is required by the CPU. This is a direct consequence of the post-cache architecture. The on-the-fly decompression ensures a maximum possible performance gain at a minimum memory requirement (since both, cache and main memory, comprise compressed code they can be smaller compared to the non-compressed case). On the down-side it requires a

[1]This implies the CPU is un-altered, processing regular uncompressed code even though the rest of the system like caches, main memory, buses etc. might process/transfer compressed code.

careful design of the decompression engine since it hosts the critical path and may actually result in an increased clock cycle time. We have solved this problem by a patented *decoding tree* hardware (not part of this paper).

3.3 Contribution and Novelties in CoCo

The following characteristics of CoCo represent the major contributions of this paper:

a) The CoCo hardware/software platform is not limited to a specific compression scheme. Rather, it can use different simple dictionary techniques provided that they can fit in reasonable space and their contents do not change during decompression.
b) The software flow is designed to be applied to a standard compilation flow without altering it. In particular, an already compiled code can be used as is and be prepared for code compression. This guarantees the application of our technology to legacy code.
c) We solve the problem of unknown (at compile time) jump and call targets. Hence, our techniques do not impose any limitations on the software such as, for example, prohibiting[2] the use of jump and call to register-stored targets, a problem that has not been addressed by others.
d) the software flow of CoCo identifies the *working set* (code segments that will be executed most often). Thus, it maximizes the performance benefits of compression under area constraints (size of dictionary; PC to CPC translation tables etc).

4. COCO HARDWARE/SOFTWARE PLATFORM

Code compression in an embedded system is more than just applying a compression algorithm to a bit stream and then using the inverse of the compression algorithm to decompress when the system is running. Indeed, technical obstacles and their solutions will decide whether the possible compression ratios can result in any gain at all, considering the overhead involved. By means of two compression schemes we show in this section the technical feasibility of code compression along with solutions to problems that have not been solved by others before. The two schemes also demonstrate the flexibility of CoCo to implement diverse solutions.
Note that the two exemplary schemes are both working with a dictionary-based compression algorithm. We do not focus in this paper on any specific compression algorithm since our CoCo platform is flexible enough to implement various table-based schemes. The two exemplary schemes are selected to showcase code-related obstacles that we have solved in CoCo.

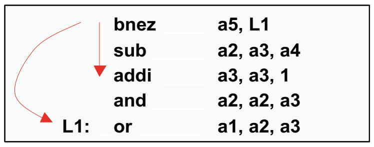
4.1 Instruction-based Compression

The granularity (see Sec. 3.1) for compression within this scheme are full instructions. There is no limitation on the instruction set architecture. Among others, the following cases are covered:

a) the instruction set can consist of instructions with different widths like, for example, Tensilica's Xtensa processor with 24-bit and 16-bit instructions.
b) the instruction-width does not have to match the bus width through which the instructions are fetched.

[2]which, in effect, may decrease the software efficiency i.e. performance

I. Deducing the Compressed Program Counter CPC
The code in Fig. 2 shows a branch instruction that, if taken, may not

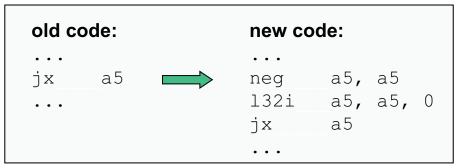


```
       bnez    a5, L1
       sub     a2, a3, a4
       addi    a3, a3, 1
       and     a2, a2, a3
L1:    or      a1, a2, a3
```

Figure 2: Problem of deducing the program counter in the decompression engine during branching in a pipelined architecture

be detected as a branch by the DCE (Decompression Engine) since the PC of the CPU continues to be incremented through a *delayed branch*. Thus, the CPC (see 3.1) cannot be deduced by the PC. A table-based translation from PC to CPC becomes necessary but would be too costly if *every* instruction address had to be translated by that table.
As a low-overhead solution, we use what we call a memory block table MBT that maps uncompressed addresses of cache blocks (not words) to compressed cache block addresses. The exact byte position of that address in the cache block is what we call the *offset* which is derived from the software itself or, in the case of indirect branch/jump/call instructions, from the main memory.
This technique is key since it saves large memory space that would be necessary for PC to CPC translation tables, which would eventually reverse the gains achieved by code compression. Here are the techniques:

1) Sequential Code: Offsets are derived from the decompression history which is a mechanism used by the DCE to track compressed instruction sizes and thus compute the next compressed address in compressed space.
2) branch/jump/calls where targets are not known at compile time: Fig. 3 shows the case of an unknown jump tar-

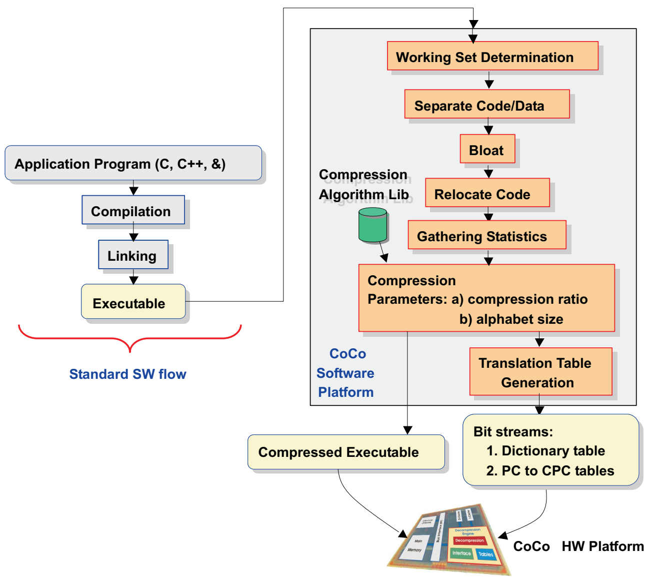


```
old code:          new code:
...                ...
jx    a5     ==>   neg    a5, a5
...                l32i   a5, a5, 0
                   jx     a5
                   ...
```

Figure 3: Software transformation to allow jumps to indirect addresses

get. Note, that the target address cannot be retrieved by a data flow analysis on the code since it is only calculated during run-time. The DCE, however, needs to know the *offset* to be able to fetch the next instruction in the compressed space. Our technique: the *neg* instruction inverts the address and then the load instruction is executed on that (invalid) address which is trapped by the DCE hardware. Then, the DCE deduces the necessary *offset* from that target address.

This is one major contribution of our work; we allow any kind of compiler-generated code without restrictions. Note though, that the code size increases through the transformations presented above.
II. Software Flow
Whereas the entire software flow of the CoCo platform is shown in Fig. 4, below are the steps that are applicable to this scheme (instruction-based compression):



Application Program (C, C++, &)
Compilation
Linking
Executable
Standard SW flow
Working Set Determination
Separate Code/Data
Bloat
Compression Algorithm Lib
Relocate Code
Gathering Statistics
Compression Parameters: a) compression ratio b) alphabet size
CoCo Software Platform
Translation Table Generation
Compressed Executable
Bit streams: 1. Dictionary table 2. PC to CPC tables
CoCo HW Platform

Figure 4: The software part of the CoCo platform

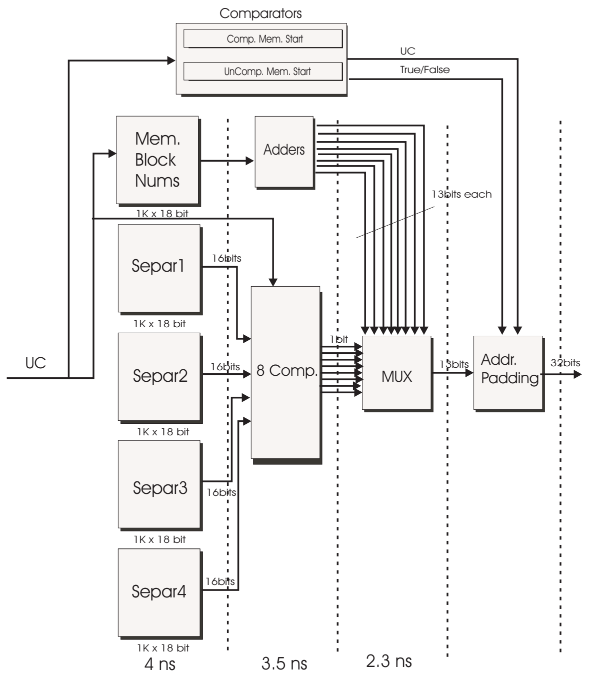


Figure 5: UC **(i.e.** PC**) to** CPC **address generation block diagram**

1) **separating code from data:** The executable is parsed and the data sections are identified and marked to avoid compression of data (which can even be located in *code* sections when using certain compilers).
2) **bloating:** Software transformations necessary to facilitate address mappings between uncompressed and compressed space (see above).
3) **gathering statistics:** Statistics are collected to obtain the frequency of appearances of instructions in the program for an efficient compression.
4) **compressing:** This phase parses the program a second time and scans for re-occurrences of instructions or instruction pairs. When found, it replaces it with an index to the compressed PC-to-CPC-table. Various table-based compression schemes can be selected by the user.
5) **generating translation tables:** This phase parses the original and the compressed program simultaneously, and generates an address mapping from uncompressed addresses to compressed ones.

III. Hardware for Addressing

The following describes the hardware within the CoCo platform to obtain addresses in the compressed space. Here are the constraints:

- the platform is capable of handling applications of any size, though the compressed regions of a program together are limited to 256KB. We call these regions the *Extended Working Set* (EWS) which is determined by software analysis (mainly profiling).
- blocks in uncompressed space (we will refer to them as UC blocks) are 256 bytes in size.
- as cache line we use mostly 32 bytes. However the platform architecture is flexible enough to handle other cache block sizes.

From the UC block size (256 bytes) and the cache block size (32 bytes) it follows that 8 separating values (we will refer to them as separators) are needed to deduce the cache block number To store these separators we use memories of 1Kx18. Four such memories are necessary each holding 2 separators. Due to a UC block size of 256 bytes, 8 bits per separator are required. Fig. 5 illustrates the block diagram of the architecture. The 13 bits of the table giving the cache block number in the EWS have to be padded with the appropriate bits depending on the location of compressed code in memory to form a full 32-bit address. The decompression history offset is concatenated to the LSB part. Thus, 15 bits are needed to be padded on the MSB side and 5 bits to be padded on the LSB side to form a full 32-bit address for the cache/memory. The main advantage of this approach is complete independence from the cache size on the architecture. A number of registers are used to make the hardware platform as general as possible:

- A register holding the number of bits in the memory block table (in the above example it is 12)
- A separator mask register: the maximum supported is 28 bits of separator storage per table entry.
- A register storing the number of separators in the separator table.

An implementation of CoCo using the Xtensa 1040 processor resulted in 90k gates on an FPGA (Altera EP20K-1500E) board running at 40 Mhz.

4.2 Word-based Compression

The granularity (see Sec. 3.1) for compression within this scheme is a word transmitted on the bus (typically 32-bit), irrespective of the actual size of the instruction which could be different. The compression metric used is described in Sec. 5.

I. Software Flow

The software flow is similar to the one in Sec. 4.1 except for: a) no modifications to the software are necessary since indexing can simply be deduced due to the constant size of an entity (bus word) that is being compressed, and b) the step ”gathering statistics” searches for occurrences of a sequence of bus words, not instructions.

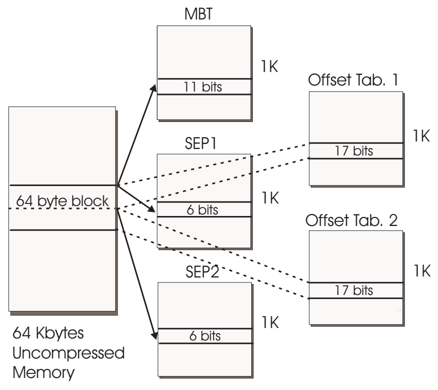


Figure 6: **Address lookup block diagram for word-based compression**

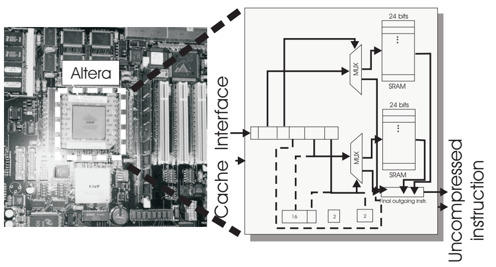


Figure 7: **CoCo's hardware platform with DCE block diagram**

II. Hardware for Addressing
Again, we have a technique to deduce the CPC without requiring a full (large) table for each word in the program space. In this scheme, CoCo limits the EWS size to 64KB. The mapping consists of a memory block table MBT (see Fig. 6) that generates the compressed memory cache block address from the uncompressed address whereas SEP1/SEP2 take care of the non-covered cache blocks (according to the compression of a word, the address of the first word is stored in MBT; the succeeding 1-2 other words are addressed via separators). Then, the offset tables determine the exact byte position in the cache block[3]. Eventually, we need 17 bits to map any address in the 64KB segment that can be located anywhere in the 4GB total address space.

5. EXPERIMENTS AND RESULTS

In the following, we present experiments using our CoCo platform. We first describe our basic assumptions, then give synthesis results (the hardware platform of CoCo is based on a board with an Altera FPGA) and we finally present compression and performance results for both the word-based technique and the instruction-based technique. The aims of the experiments were to demonstrate:

a) the ability of the platform to run different code compression techniques. We deployed a reconfigurable processor with an on-chip cache that allowed us to modify cache parameters such as the cache size,
b) the ability of CoCo to load/execute diverse (size, application domain etc.) applications. Our CoCo software platform takes any executable and patches all necessary information for execution (like space for the decompression memory block and offset tables). During execution the processor loads first the tables into the SRAM(s) and then starts executing the on-the-fly decompressed application,
c) the ability of our method to handle complex instruction sets with variable instruction lengths. Therefore, we used a reconfigurable processor that has two different instructions sizes (24-bit and 16-bit) instructions. The processor was configured to run on a 32-bit instruction bus. Thus, at any time a 32-bit word is received and has to be processed by the decompression unit DCE,
d) to evaluate our complex software package that accomplishes code transformations (an example was given in 4.1).

[3]The offset tree structure is left out since it is currently being filed as a patent

The hardware platform of CoCo using an Altera FPGA is shown in Fig. 7 along with a block diagram of the DCE.

The timing and area results for "instruction-based"/"word-based" methods are: $28ns/7.2ns$ clock cycle time ($\approx 40MHz/130MHz$) on a 0.25μ[4] while the area usage equals to 11K/2K gates for the decompression logic and 16.75K/15.5K for the SRAMs containing the address mapping tables.

5.1 Compression and Performance Results

The first series of results stems from our instruction-based compression scheme (see 4.1). They are based on two levels of compression: the first level involves identifying the 1024 (i.e. size of the lookup table on the hardware platform) most frequently appearing 24-bit instructions and compressing them into 16 bits, while the second level involves scanning for three frequently appearing consecutive 16-bit instructions (i.e. 48 bits in total) and compressing them into 16 bits. The amount of compression is limited by our table sizes: our platform has space for 1024 24-to-16 bit compressed instructions and 1024 48-to-16 compressed instructions.

The second series of experiments are based on "word-based" compression (see 4.2). In this case, our tables can accommodate 1024 64-to-32 bit compressed instructions. In other words, the 1024 most frequently appearing pairs of words are grouped and are compressed into *one* word (32-bit data bus). We used in all experiments a 32-bit wide address bus (this is the maximum address space of CoCo and as such requires the highest bit width for the address mapping tables; see 4.1 and 4.2). Furthermore, the cache line sizes were 32 bits throughout.

The applications are: *advance* (3100 lines of C code, 188KB executable), a math calculation application that makes use of complex jump instructions (through switch statements) and *smo* (1245 lines of C code, 74KB executable): an image smoothing application that is dominated by bit operations.

Table 2 shows compression and performance results for our two applications using the "word-based" technique. It shows various cache sizes in the compressed an un-compressed case, the number of times a cache line is refilled, the total number of cycles that were cache hits *hit cycles*, the total number of cycles in the application *total cycles* and finally the cycles through compression. For the *advance* application we observe that smaller caches show larger benefits as the execution pattern of the application is concentrated to a small fraction of the code such that larger cache sizes do not result in further improvements. This execution pattern (through the EWS metric) leads to a partially full dictionary such that the upper bound

[4]Note that because of the prototyping nature of our platform we need not to deploy the newest silicon technology. In fact, this older technology enables us processors on our platform that are not yet available in the newest silicon technology.

Appl.	CR	Cache	Cache	Hit	Total	Cycles
Appl.	CR	size	Refills	cycles	cycles	saved
adv	no comp	1k	1,538	354,813	388,649	n/a
	75%	1k	1,220	354,020	380,860	7,789
	no comp	2k	1,270	354,713	382,653	n/a
	75%	2k	1,002	353,745	375,789	6,864
	no comp	4k	1,104	354,652	378,940	n/a
	75%	4k	836	353,696	372,088	6,852
	no comp	8k	994	354,610	376,478	n/a
	75%	8k	752	354,286	370,830	5,648
	no comp	16k	955	354,592	375,602	n/a
	75%	16k	731	353,979	370,061	5,541
smo	no comp	1k	53,855	3,879,327	5,064,137	n/a
	77%	1k	52,140	3,880,928	5,028,008	36,129
	no comp	2k	35,152	3,866,202	4,639,546	n/a
	77%	2k	33,949	3,854,456	4,601,334	38,212
	no comp	4k	7,739	3,867,419	4,037,677	n/a
	77%	4k	4153	3,867,729	3,959,095	78,582
	no comp	8k	53	3,863,575	3,864,741	n/a
	77%	8k	43	3,863,586	3,864,532	209

Table 1: "instruction-based" (see 4.1) compression: Results in terms of compression ratio and performance for both, compressed and uncompressed instruction code for various instruction cache sizes

Appl.	CR	Cache	Cache	Hit	Total	Cycles
Appl.	CR	size	Refills	cycles	cycles	saved
adv	no comp	1k	1,538	354,813	388,649	n/a
	50%	1k	863	354,085	373,071	15,578
	no comp	2k	1,270	354,713	382,653	n/a
	50%	2k	672	354,030	368,814	13,839
	no comp	4k	1,104	354,652	378,940	n/a
	50%	4k	546	353,399	366,011	12,929
	no comp	8k	994	354,610	376,478	n/a
	50%	8k	502	353,983	365,027	11,451
	no comp	16k	955	354,592	375,602	n/a
	50%	16k	463	354,241	364,427	11,175
smo	no comp	1k	53,855	3,879,327	5,064,137	n/a
	69%	1k	50,126	3,887,107	4,989,879	74,258
	no comp	2k	35,152	3,866,202	4,639,546	n/a
	69%	2k	30,911	3,878,879	4,558,921	80,625
	no comp	4k	7,739	3,867,419	4,037,677	n/a
	69%	4k	43	3,863,566	3,864,512	173,165
	no comp	8k	53	3,863,575	3,864,741	n/a
	69%	8k	43	3,863,565	3,864,511	230

Table 2: "word-based" (see 4.2) compression: Results in terms of compression ratio and performance for both, compressed and uncompressed instruction code for various instruction cache sizes

of compression i.e. 50% can be achieved whereas *smo* utilizes the whole dictionary and in fact not all patterns could be stored, effectively resulting in a 69% compression ratio. The performance is improved in all cases through the effect of an effectively higher bus bandwidth and through an effectively larger cache (since it holds compressed instructions. "Instruction-based" Table 1 shows similar results; it depends on the application which technique achieves better results.

The current limitation of the platform is that we can only deploy simple dictionary-based compression methods. But, as mentioned earlier, only dictionary/table-based techniques are capable to decompress on-the-fly under a feasible hardware effort and without adding extra cycles.

6. SUMMARY AND CONCLUSION

We introduced in this paper our CoCo hardware/software platform that allows to rapidly prototype and evaluate diverse code compression technologies. It is the first such platform of its kind. We have demonstrated the flexibility of CoCo by showcasing two dictionary/table-based compression schemes ("instruction-based" and "word-based"). CoCo has no restrictions on the code (like all other known approaches) since it allows to compress *any* instruction of an instruction set architecture (including jump instructions to register-stored targets). It can handle instructions set architectures that feature multiple instruction word lengths. Furthermore, we have shown that CoCo can implement any dictionary/table based compression algorithm. As a result, CoCo allows to rapidly prototype diverse techniques and, among others, to trade-off compression ratios against performance.

7. REFERENCES

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. 23(No 3):pp 337–343, May 1977.

[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. Xie. Code Compression Algorithms and Architectures for Embedded Systems. *PhD Thesis*, Princeton University 2002.

[11] 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.