Multi-Profile Based Code Compression

E. Wanderley Netto
CEFET/RN IC/UNICAMP

R. Azevedo
IC/UNICAMP

P. Centoducatte
IC/UNICAMP

G. Araujo
IC/UNICAMP

Caixa Postal 6176
13084-971 Campinas/SP Brazil
+55 19 3788 5838

{braulio, rodolfo, ducatte, guido}@ic.unicamp.br

ABSTRACT

Code compression has been shown to be an effective technique to reduce code size in memory constrained embedded systems. It has also been used as a way to increase cache hit ratio, thus reducing power consumption and improving performance. This paper proposes an approach to mix static/dynamic instruction profiling in dictionary construction, so as to best exploit trade-offs in compression ratio/performance. Compressed instructions are stored as variable-size indices into fixed-size codewords, eliminating compressed code misalignments. Experimental results, using the Leon (SPARCv8) processor and a program mix from MiBench and Mediabench, show that our approach halves the number of cache accesses and power consumption while produces compression ratios as low as 56%.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; C.3 [**Special Purpose and Aplication Based Systems**]: Real-time and Embedded systems.

General Terms

Performance, Design.

Keywords

Code compression, compression, code density.

1. INTRODUCTION

Embedded computing has been moving toward sophisticated high end systems. As a result, embedded programs are becoming larger, often requiring high-performance processors to meet performance constraints. RISC processors have been traditionally used to integrate the core computational unit of high-end embedded systems. However, memory area is one of the most constrained resources in embedded systems and code density is not RISC best feature.

One of the possible solutions to squeeze code is the usage of code compression. Unfortunately, special program requirements like the possibility of starting decompressing at any point in the code (or at least at basic blocks entries) discard some outstanding data compression algorithms to be directly applied. Another requirement is the ability to decompress an instruction at run time without prohibitively affecting performance.

Since the Compressed Code RISC Processor – CCRP [15] was introduced, many compression techniques have been shown to be efficient in code compression [1,3,11] and a few have been adopted by industry [7,12]. Recently, researchers have realized that the benefits of compression go beyond reducing code size, reaching performance improvement and energy consumption reduction [2,10].

This becomes evident when the decompressor engine is positioned between the processor and the cache – a Processor-Decompressor-Cache (PDC) architecture. In this scheme the cache holds compressed instructions, thus increasing its capacity and reducing misses. This reduction promotes less main memory accesses, saving energy and clock cycles. However, the decompressor engine can considerably impact performance because it is placed on the processor critical path. If the decompressor impact is small, the net result is an improvement on performance and reduction in energy consumption. Fast decompression techniques, like those based on small dictionaries [2,10], have been used to reduce this impact.

Since indices into the dictionary are usually smaller than an instruction word, the compressed code stream gets misaligned, frequently requiring two accesses to the cache to decompress one instruction. Moreover, instructions redundancies may occur when using dictionaries with multiple instructions per entry, because their sequence must be obeyed. These restrictions downgrade some benefits on energy consumption and performance.

In the present work we propose a dictionary-based technique that packs a set of indices into a fixed size 32-bit word called *ComPacket*. This regular size word allows intermixing original instructions with ComPackets, thus eliminating the double cache accesses and alignment problems mentioned above. Besides that, at a cost of one 32-bit buffer, we hold this word inside the decompression engine, thus avoiding unnecessary cache accesses.

Finally, we have noticed that the dictionary construction considerably impacts the final code compressibility and dynamic behavior. When static instruction count is elected as the approach to select instructions to the dictionary, it usually yields better compression results and some performance benefits. On the other hand, performance improvements may be considerably enhanced by choosing instructions based on dynamic profiling information. We propose a mix of static/dynamic instruction count selection criteria to construct one dictionary that performs well for both scenarios.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC'04, June 7–11, 2004, San Diego, California, USA
Copyright 2004 ACM 1-58113-828-8/04/0006…$5.00.

On average, our results point to a reduction of 35% in code size when using only static information to build the dictionary, and a reduction in cache accesses of 53% when using just dynamic profiling. Cache energy consumption is also reduced to 54% of the original value.

In this paper, Section 2 presents the background and related work on code compression. Section 3 explains the dictionary construction. Section 4 presents the compression method. In Section 5 the experimental results are shown and Section 6 presents our conclusions.

2. BACKGROUND AND RELATED WORK

The metric widely used to measure the code compression efficiency is the compression ratio (CR). When using dictionary techniques, the dictionary itself becomes part of the compressed object code (whenever for each application we find one specific dictionary). To make this explicit, we include the dictionary size in Equation 1 that we use to measure compression ratio. Notice that the lower the ratio, the better the compression.

$$CR = \frac{Compressed\ Program\ Size + Dictionary\ Size}{Original\ Program\ Size} \qquad \text{Eq. 1}$$

One of the key challenges in code compression comes from the address misalignment between original and compressed programs. This requires an efficient address translation function to map addresses during decompression. Two possible solutions to this problem are well known: using Address Translation Tables (ATT) [15] and patching the addresses in the compressed code [9]. Patching is more suitable to PDC architectures because an ATT would require a one-to-one address translation.

2.1 Related Work

Lefurgy *et al* [9] experiments used fixed and variable-length codewords[1]. In their first method, 16-bit fixed-length codewords are used with a 256-entry dictionary, each entry containing up to 4 instructions in sequence. Their approach to variable-length codewords uses a prefix of four bits to determine the size of the codeword. The code stream comes in chunks of 32 bits, which may contain partial instructions and/or codewords. Their best results produced compression ratios of 61%, 66% and 74% for the PowerPC, ARM and i386, respectively. No report on energy and performance is available.

Benini *et al* [2] used a dictionary based compression method formed by using dynamic instruction profiling information. A small 256-entry dictionary with one instruction per entry is used to keep the most executed instructions. They compress instructions that belong to the dictionary if they can be accommodated in a cache line size package. For every compressed cache line, the first 32-bit word is used as an escape sequence and a set of flags to indicate the presence of compressed/ uncompressed instruction in the remaining 12 bytes of the line. A 75% compression ratio is produced for the DLX. Also, a 30% energy reduction was obtained. Unfortunately, cache access time is increased in 32% as the decompressor is coupled with the cache.

Lekatsas *et al* [10] used the Xtensa 1040 processor to support a small dictionary compression method based on static instruction count. This 32-bit processor has irregular instructions sizes of 16 and 24 bits. The authors used variable-length codewords of 8 and 16 bits to compress the original 24 bit instructions. Their primary goal was to guarantee that the decompressor engine requires no more than one cycle to decompress one or two codewords. Some decompression overhead comes from the fact that the engine is supposed to keep fractions of misaligned instructions or codewords that come from the cache. Moreover, the dictionary was doubled (2x256-entry) to support two codeword decompression per cycle. A 35% code size reduction was achieved and a 25% performance improvement (cycles count reduction) was reported. Unfortunately, no detailed information about memory hierarchy parameters (like cache miss penalty) is available to compare to our work.

Our compression method differs from previous work by the use of word-sized sets of indices, eliminating compressed code misalignments, and by our new approach to build the dictionary, based on mixing static and dynamic profiling.

3. DICTIONARY CONSTRUCTION

The composition of a small dictionary is based on any classification of instructions or pieces of instructions. One widely used classification is the static occurrence of every instruction in the code. Whenever compression ratio is the main goal, this approach yields the best results, as we represent with fewer bits the instructions that appear the most. Other methods use dynamic profiling information to guide the dictionary construction. This tends to put into the dictionary the most executed instructions, and may be very effective in some dynamic goal, like bus toggles minimization. We use in this paper the name *static dictionary (SD)* for those dictionaries built upon static classification of instructions. Similarly, we name *dynamic dictionary (DD)* those formed by the execution count of every instruction.

The question we have been investigating is: how different are SD and DD? We restrict the answer to the case of small dictionaries with one instruction per entry, as they are appropriate to the PDC architecture we are investigating. Figure 1 shows the number of redundant instructions (intersection of SD and DD) for several dictionary sizes. Notice that, for our case (256-entry dictionary) 30% of instructions are redundant on average.

Figure 2 shows how skewed is instruction count distribution inside the dictionary. We ranked and normalized the values to the biggest count obtained. We observe that the first instructions in the dictionary determines much of the compressibility. The dynamic dictionary has a very similar behavior. Observe that the x-scale in the graph is not linear to help in the visualization of its leftmost part.

From these observations we outline a dictionary composition that contains the most static occurred instruction, as well as, the most executed ones. The assumption is that by choosing the instruction which statically appears the most will help in the compression ratio, and at the same time, choosing the instruction that is fetched the most will help in performance.

To blend SD and DD into an Unified Dictionary (UniD) we begin by ordering them by their natural selection criteria: static count for the SD and dynamic count for the DD. Then we take the first (most occurred) instruction from SD and check if it is present in UniD. If not, we include it. We do the same to include one instruction from DD, the second from SD, the second from DD, and so on. We finish searching instructions from SD and DD when the UniD is full. This simple version is described in [13].

An extension to this basic approach is the possibility of choosing a threshold for 'dynamic instructions' in UniD. We can vary the DD instruction proportion in UniD from none (0%) to all entries (100%). We call this the dynamic factor, *f*, of the UniD.

[1] A codeword is the bit pattern attributed to an index.

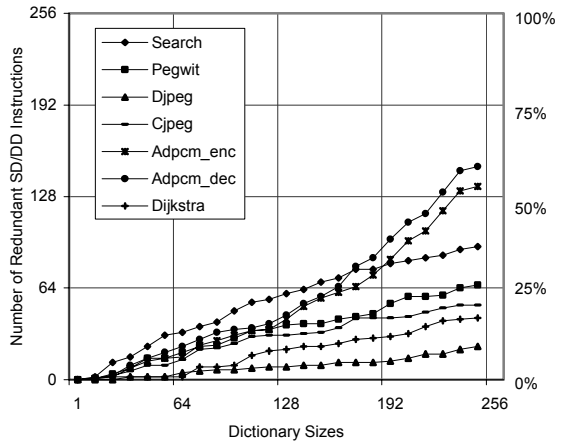

Figure 1: Dynamic vs. Static Dictionaries Similarity

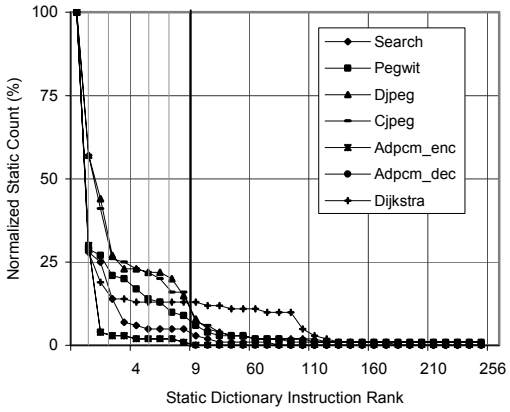

Figure 2: Dictionary Instruction Static Usage Count

Eq. 2 defines *f*.

$$f = \frac{Instructions\ from\ DD}{Dictionary\ Size} \times 100\% \qquad \text{Eq. 2}$$

As a result, we can exploit the dictionary composition space to meet special system requirements. Of course when choosing f = 0%, some instructions from the DD still belong to UniD because there is an intersection between the SD and DD sets. The dictionary construction algorithm guarantees that at least ($f \times$ |UniD|) instructions come from DD.

This dictionary construction method opens up opportunities to investigate the behavior of the code compression algorithm in several situations. As an example, Figure 2 presents the curve we obtain when computing the CR and bus[2] toggle ratio (compressed over original program bus toggles), for the pegwit program (avg. case). The best compression ratio can be obtained when f=0%. On the other hand, minimization of bus toggles is best explored when the entire dictionary is formed by instructions from DD (f=100%).

The compression ratio evolution is smoother than the bus toggle ratio, but notice that when f is greater than 90%, a typical knee in the curve is formed. On the other hand, from f=0% to f=20% an expressive reduction in bus toggles is achieved.

[2] Between the decompressor and the I-cache.

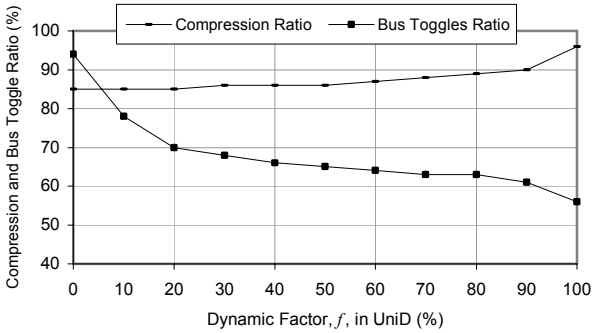

Figure 3: Typical Trade-off Exploitation (Pegwit program example)

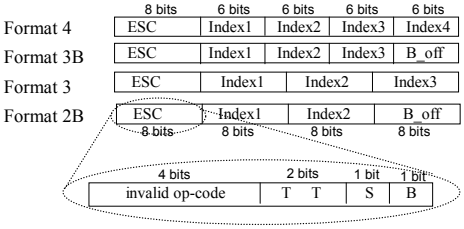

Figure 4: ComPackets Formats

We conclude that relying only in one statistic (either static or dynamic profiling) is less effective than relying on both at the same time.

4. COMPRESSION METHOD

To explore the functionality of our dictionary construction method we devised a technique that uses some ideas on the state-of-the-art PDC code compressions schemes but with original solutions to their weakness and a new coding approach.

Different from [9], our compression method allows relative branches to belong to the dictionary if they have a small offset (the original 22-bit displacement can be represented with only 8 bits). In fact, small branches are the majority in typical program. In order to cope with the offset patching problem in small branches that belong to the dictionary, we keep just the 10 most significant bits of branch instructions in the dictionary and use a slot of 8 bits inside the ComPacket to store the patched offset.

We also allow targets to be any index, so that whenever branching into a ComPacket the target is not necessarily the first index, reducing the required alignment padding from other methods [2,10]. We currently do not support two or more targets (or branches) in the same ComPacket.

As aforementioned, we use a 32-bit word as a repository for sets of indices. An escape sequence of 8 bits is used at a fixed position, allowing fast decoding and interpretation of the remaining 24 bits. This escape sequence is responsible to differ the ComPacket word from an uncompressed instruction and it also includes information about the number of indices and its sizes, the presence of small branch offset and target index.

Figure 4 presents the four ComPackets formats our compression method supports. They are named after their contents. Format 4 has 4 indices of 6 bits into the dictionary such that just the first 64 entries are accessible. Format 3 has 3 indices of 8 bits each, thus allowing full access to the 256 entries of the dictionary we use. Format 3B has 3 indices and branch slot of 6 bits. This format restricts the branching offset size, so that just tiny branches (22-bit displacement that can be represented with 6

```
Compress()
  1.  Build the Dictionary (Section 3)
  2.  Code Marking
  3.  Find dictionary instructions in the Code
        a.  Try to mark Format 4 ComPacket
        b.  Try to mark Format 3/3B ComPacket
        c.  Try to mark Format 2B ComPacket
  4.  Assembly ComPacket formats marked in 3.
  5.  Replace ComPackets in the code
  6.  Patch addresses
```

Figure 5: Compression Algorithm Outlined

bits) are allowed. Format 2B is a repository for 2 8-bit indices with possibly one of them being a branch. In this case, the last 8 bit slot is filled with the small branch offset.

The identification of the format is done by a pair of bits (S and B) in the escape sequence depicted in Figure 4. Whenever S = 0, the ComPacket uses slots of 6 bits for indices and/or branches offsets. Whenever S=1 the slots are 8 bit-long. The B bit signals the presence of a slot containing a branch offset. Notice that the branch offset is always in the last slot independent of which index in the ComPacket points to a branch. An extra bit in front of each dictionary entry identifies this kind of instruction. For the 256 entries dictionary, this extra bit represents just a 3% overhead in its bit capacity.

The TT pair of bits points to the index from which execution should begin after a branch into the ComPacket. When TT = 00_2 the target is the first index, when TT = 01_2, the second, and so on. If ComPacket is entered sequentially (not due to a branch), the first index is always used. The remaining 4 bits of the escape sequence are used for assigning an invalid instruction in the SPARC ISA (op = 00_2 and op2 = $00X_2$). Notice that Figure 4 is a pictorial representation of the bit fields. They are actually disjoint but have a fixed location.

4.1. Compression Algorithm

The algorithm for compression is outlined in Figure 5. After the dictionary construction (stated in Section 3) it marks in the code the instructions that belong to the dictionary (D) and/or are targets (T). Then, it tries to select and mark ComPacket formats for each set of instruction in dictionary, from the most effective (Format 4) to the simpler (Format 2B). After choosing the formats, the compressor assembles and replaces them in the code. Finally, the branch addresses are patched.

Figure 6 shows the algorithm in action. After the dictionary construction and marking steps, it considers the first instruction in

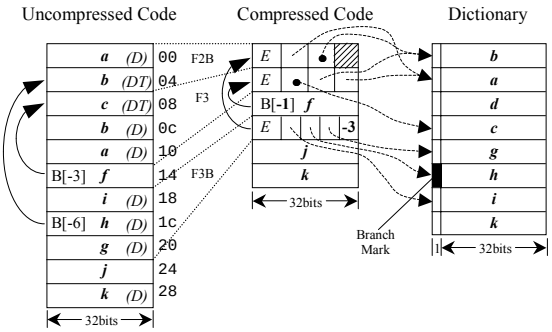

Figure 6: Compression Example

the dictionary: *b*. It finds in the code the first instance of *b* (at address `04`) and tries to build a Format 4 ComPacket. Although *a*, *b* and *c* are dictionary instructions, they cannot compose a Format 4 (`00` to `0c` addresses) because two instructions are targets. Format 3 or 3B are not possible for the same reason. Format 2B is possible by combining *a* and *b*, thus they are marked compressed. As they are not branches the last slot is padded (hachured in figure). The second instance of *b* is at address `0c` and again a Format 4 is not possible to be formed because just *c*, *b* and *a* (at addresses `08` to `10`) are still uncompressed (*b* at address `04` was marked compressed in the former step). In this case a Format 3 is formed, as no branch instruction is present. As it finishes with *b* the algorithm considers the next instructions in the dictionary. The formats are presented in the figure bounded by dotted lines.

Next, the formats chosen are applied to the code. See in the Compressed Code the dashed lines pointing into the dictionary entries. Finally branch addresses are patched. Instruction *f* is patched normally inside the new code. Instruction *h* is patched inside the ComPacket. Notice the branch mark in the corresponding dictionary entry, the target marks (dots) and offsets in the compressed code.

5. RESULTS

We used a simulator of the Leon processor (SPARC v8) [4] developed in our lab to support our experiments. The benchmarks are extracted from MiBench [5] and Mediabench [8]. They are a string search algorithm, *Search*, commonly used in office suites; *Dijkstra*, an algorithm used in network routers; *Djpeg* and *Cjpeg*, used for compressing and decompressing images from and to JPEG formats; *Adpcm* encodes or decodes audio; and *Pegwit*, an encryption tool.

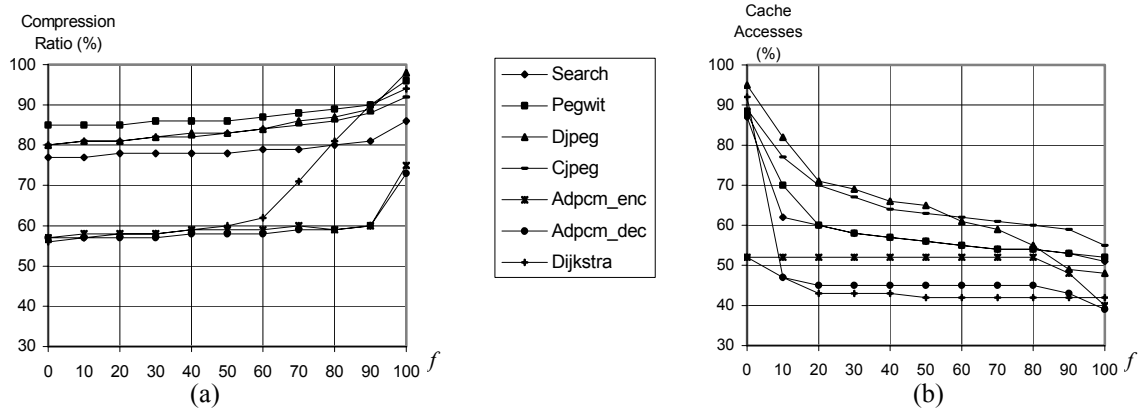

Figure 7: (a) Compression Ratio and (b) Accesses Reduction to the I-Cache

We used LECCS, a GCC based compiler for the Leon processor, with –O2 option in all the benchmarks, so that we avoid typical optimizations that increase object code size, like function in-lining and loop unrolling.

Figure 7(a) depicts the compression ratio for the complete range of experiments. They exploit all compositions of dictionaries from static and dynamic measurements. The best compression, as expected, is obtained by using only instructions from SD (f = 0%). Nevertheless, by using some combination in which f is lower than 50%, the impact on compression ratio is quite small. This is not the case when f goes up to 90% so that compression is severely affected by the dictionary composition.

The claim of using the regular word-sized ComPacket is to avoid more than one access to the cache to decompress one instruction. Moreover just one cache lookup is necessary to execute the entire set of instructions pointed by the indices inside the ComPacket (as it is kept in a decompressor engine's buffer).

Figure 7(b) presents the results on cache access reduction. Notice that accesses strongly decrease from f=0% to f=20%. After that, an incremental decrease is observed until f=90%, where decrease is again enhanced (although not like the first 20%). As f grows, more ComPackets are found during execution providing more accesses reduction. This kind of figure can be used to help choosing an ideal proportion of instructions in the dictionary to meet specific system requirements.

We also ran experiments to measure the bus activity between the decompressor and the cache. This represents an important metric to define bus energy consumption. We were particularly interested in measuring the Hamming distance accumulated through the execution of the code. In Table 1 we present the net results obtained by adding the addresses and code bit toggles for four selected values of f: 0%, 20%, 50% and 100%.

These values were chosen because, as aforementioned, when f gets near 20% a significant decrease in bus toggles is already sensed. One exception in this behavior is the Adpcm, for which a good reduction in bus toggles is observed, even when using a static dictionary. We attribute this behavior to two factors: the size of the original code (less than 2K instructions), and the greatest similarity between SD and DD that we measured.

Table 1: Bus Toggles Reduction (% of original)

	Original (uncomp)	***f***			
		0%	**20%**	**50%**	**100%**
Search	112,028,630	90%	65%	60%	54%
Pegwit	409,946,579	94%	70%	65%	56%
Djpeg	45,878,397	97%	78%	74%	54%
Cjpeg	194,398,718	96%	78%	73%	63%
Adpcm_enc	126,567,600	63%	58%	56%	45%
Adpcm_dec	96,667,537	53%	48%	48%	42%
Dijkstra	665,188,662	92%	56%	52%	49%
	Average	***84%***	***65%***	***61%***	***52%***

5.1 A Selected Case of *f*

Finally, we chose a dictionary composition to show how it is related to the best and worst results. The value of f is fixed in 50% and we trade Compression for Cache Accesses. Table 2 summarizes the results. We see that such a dictionary performs very close to the best compression ratio achieved, differing only 2% on average. It also substantially differs from the worst compression ratio in 15%. Furthermore, the cache reduction is 7% worse than the best results. This contrasts with the 25% distance from the worst case. Such a dictionary approaches the best results from both scenarios at the same time. Thus by mixing instructions from the SD and the DD we can have high compression ratios with lesser caches accesses, reducing energy consumption.

One important aspect of compression, especially when a PDC architecture is used, is the cycle overhead the decompressor may produce. Using small dictionaries usually satisfies the cycle time budget available [10] for decompression in one cycle. Nevertheless, as the decompression engine is positioned like a pipeline pre-fetch stage, at least one cycle overhead is observed when a branch is taken. In our scheme this extra cycle is just required whenever a branch is taken to outside the current ComPacket. The claim is that this extra-cycle is compensated by the reduction in cache misses.

In fact, any reduction in cycle count depends on cache size and miss penalty. To demonstrate our experiments we have chosen for each benchmark a representative cache size, around the point in which an increase in cache size produces just a modest hit ratio improvement. They are all direct mapped with 16 bytes per line.

Then, we have explored the final cycle count as a function of the cache miss penalty. Miss penalty is much dependent on the memory hierarchy (a second level of cache, an on-chip RAM, an off-chip flash, their size and technology). Hennessy and Patterson point somewhat from 8 to 150 penalty cycles for a L1 cache miss [6].

In Figure 8 we present the results from a hypothetical 0 cycles miss penalty to 100 cycles. Whenever a cache miss has a penalty of 10 cycles, a total cycle reduction of 27% is achieved. Even for a fast main memory like this, the decompressor overhead in cycles is completely outweighed by the savings in main memory.

For the same platforms we have evaluated the energy consumption in the I-Cache. The adopted energy model was extracted from the CACTI tool [14] for a CMOS 0.8um

Table 2: Relation of Best/Worst Results for a Selected Case of *f*

	Compression Ratio					Cache Accesses Reduction				
	Δ % From Best	Δ % From Worst	Best Value	Worst Value	*f*=50%	Worst Value	Best Value	Δ % From Worst	Δ % From Best	
Search	1	9	78	86	77	56	87	51	31	5	Search
Pegwit	1	10	85	96	86	56	88	52	32	4	Pegwit
Djpeg	3	15	80	98	83	65	95	48	30	17	Djpeg
Cjpeg	3	9	80	92	83	63	89	55	26	8	Cjpeg
Adpcm_enc	2	16	57	75	59	52	52	40	0	8	Adpcm_enc
Adpcm_dec	1	15	57	73	58	45	52	39	7	6	Adpcm_dec
Dijkstra	4	34	56	94	60	42	92	42	50	0	Dijkstra
Average	*2*	*15*							*25*	*7*	*Average*

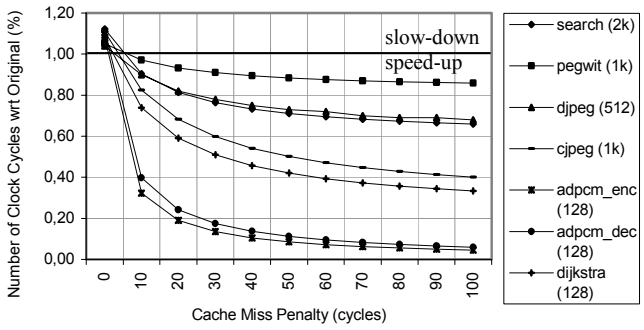

Figure 8: Execution Cycles Relative to Original

technology. Table 3 summarizes the results. It shows the energy consumed in the I-Cache without and with code compression. The percentile reduction is also presented. Notice that an outstanding energy reduction of 46% is obtained on average.

Regrettably, this energy reduction in the cache comes at a cost. The dictionary lookups also consume energy. Fortunately, this value becomes negligible when the cache size increases. For a 256k cache the dictionary consumption represents just 3% of the cache energy.

We observe that one drawback of this work is the fact that we cannot compress one instruction that belongs to the dictionary if it is not neighbor to other(s) that also belongs to the dictionary (case of *k* in Figure 6). This implies that the compression ratio could be better. On the other hand, the code stream does not have pieces of instructions, as proposed by other researches, considerably simplifying decompression.

Finally we conclude that packing indices into the regular structure of the ComPacket enhances cache accesses reduction yielding energy savings and performance improvement. Moreover, our dictionary construction method is able to approach *simultaneously* the best results from pure static and pure dynamic information based dictionaries code compression techniques which represents an advance in trade off exploration.

Table 3: I-Cache Energy Consumption

	Uncompress. code Joule x 10^{-3}	**Compressed code Joule x 10^{-3}**	**Reduction (%)**
Search	30.34	17.10	44
Pegwit	104.34	58.93	44
Djpeg	9.82	6.38	35
Cjpeg	47.68	30.26	37
Adpcm_enc	19.94	10.38	48
Adpcm_dec	14.84	6.80	54
Dijkstra	110.66	47.47	57
		Average	**46**

6. CONCLUSIONS

So far we have presented a new dictionary based compression method that is independent of the cache organization and uses a simple low-impact decompressor. The compression uses a regular 32-bit word named ComPacket that holds variable-length indices pointing into the dictionary. The decompressor holds the ComPacket until an instruction outside its boundary is required, thus avoiding unnecessary cache accesses. We also studied different dictionary construction methods to allow an effective trade-off exploration for compression, energy reduction and performance at the same time.

On average the compression ratio obtained was 70% taking into account the dictionary (that represents about 5% of the compressed code, so a 35% code size reduction was achieved). We also show a reduction of more than 50% in cache accesses, a 27% reduction in cycle count even with a fast main memory and an impressive 46% reduction in energy dissipated in the I-Cache.

7. ACKNOWLEDGMENTS

Our thanks to CNPq and FAPESP for supporting this work (grants #140631/2001-1, #552117/2002-1 and #2000/15083-9).

8. REFERENCES

[1] Araujo, G., Centoducatte, P., Azevedo, R. and Pannain, R. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE transactions on VLSI Systems 8,5 (Oct. 2000), 530-533.*

[2] Benini, L., Macci, A. and Nannarelli, A. Cached-code compression for energy minimization in embedded processor. In *Proceedings of the International Symposium on Low Power Electronics and Design (Aug.2001), 322-327.*

[3] Debray, S. and Evans, W. Profile-guided code compression. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (June 1998), 95-105.*

[4] Gaisler, G. Leon, 2003. see: www.gaisler.com

[5] Guthaus, M., Ringenberg, M., Ernst, D., Austin, T., Mudge, T. and Brown, R. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (Dec. 2001), 3-14.*

[6] Hennessy, J. and Patterson, D. *Computer architecture: a quantitative approach, 3rd ed.* Morgan Kaufmann Publ. 2002.

[7] IBM*, CodePack: PowerPC code compression utility user's manual. V3.* IBM Corporation, 1998.

[8] Lee, C., Potkonjak, M. and Mangione-Smith, W. MediaBench: a tool for evaluating and synthesizing multimedia communication system*.* In *Proceedings of the Int'l Symp. on Microarchitecture (Dec. 1997), 330-337.*

[9] Lefurgy, C., Bird, P., Chen, I-C. and Mudge, T. Improving code density using compression technique*.* In *Proc. of the Int'l Symp. on Microarchitecture (Dec. 1997), 194-203.*

[10] Lekatsas, H., Henkel, J. and Jakkula, V. Design of one-cycle decompression hardware for performance increase in embedded systems. In *Proceedings of the Design Automation Conference (June 2002), 34-39.*

[11] Lekatsas, H. and Wolf, W. SAMC: a code compression algorithm for embedded systems*. IEEE transactions on CAD 18,12 (Dec. 1999), 1689-1701.*

[12] Seal, D. *ARM Architecture Reference Manual, 2nd ed.* Adison-Wesley, Reading/MA, 2000.

[13] Wanderley Netto, E., Azevedo, R., Centoducatte, P. and Araujo, G. Mixed static/dynamic profiling for dictionary based code compression. In *Proceedings of the International Symposium on System-on-Chip (Nov. 2003), 159-163.*

[14] Wilton, S. and Jouppi, N. CACTI: An enhanced cache access and cycle time model. *IEEE J. of Solid-State Circuits 35,5 (May 1996), 677-688.*

[15] Wolfe, A. and Chanin, A. Executing compressed programs on an embedded RISC architecture*.* In *Proceedings of the Int'l Symp. on Microarchitecture (Dec. 1992), 81-91.*