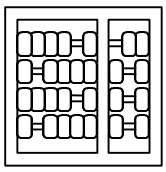

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Code Compression to Reduce Cache Accesses

Eduardo Wanderley Netto
Paulo Centoducatte
Rodolfo Azevedo
Guido Araujo

Technical Report - IC-03-023 - Relatório Técnico

November - 2003 - Novembro

Code Compression to Reduce Cache Accesses

Eduardo Wanderley Netto[*‡] Rodolfo Azevedo[‡] Paulo Centoducatte[‡] Guido Araujo[‡]

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 hits, thus reducing power consumption and improving performance. This report proposes a simple dictionary-based technique for code compression. It describes an approach to mix static/dynamic instruction profiling so as to trade-off compression ratio for performance. Compressed instructions are stored as variable-size indices into fixed-size codewords, thus reducing dictionary entry superpositions and compressed code misalignments simultaneously. Moreover, a decompressor buffering scheme is proposed that considerably decreases cache lookups. 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 while produce compression ratios as low as 56%.

1 Introduction

Memory area is one of the most constrained resources in embedded systems. On the other hand, embedded computing has been moving toward sophisticated systems that implement complex algorithms, user interfaces and real time support [1]. As a result, embedded programs are becoming larger, and high-performance processors are required to meet design constraints. RISC processors have been traditionally used to integrate the core computational unit of high-end embedded systems. Their performance and the availability of optimized compilers and IP cores make them the best design choice for such systems. However, this choice imposes an overhead in terms of code density, given that RISC code is known to be larger than their equivalent CISC and DSP counterparts.

One of the possible solutions to squeeze code is using code compression. Techniques from the data compression field have inspired researches to derive methods that can be applied to code, but unfortunately, special program requirements like the possibility of starting decompressing at any point in the code, have discarded some outstanding data compression algorithms. This is a required feature for programs that support control flow branches. Another requirement in code compression is the ability to decompress an instruction, or block of instructions, at execution time without prohibitively affecting performance.

Since the CCRP [2]was introduced, researchers have shown that the benefits of compression go beyond reducing code size, reaching energy consumption reduction and performance improvement [3,4,5,6,7]. Such improvement becomes evident when the cache holds compressed instructions, effectively increasing its capacity, thus reducing cache misses. To make it real, the decompressor engine is positioned between processor and cache – a Processor-Decompressor-Cache (PDC) architecture. However, this scheme may impact the performance, given that the decompressor engine is placed on the processor critical path.

The frequent used solution to reduce the decompressor impact is to use a dictionary based compression method where compression is achieved by replacing instruction sequences with indices into the dictionary [8]. When a dictionary is size-limited in terms of number of sequences it can hold, a selection criterion must be followed to fill its entries with the most advantageous sequences. Such selection is based on static instruction count [5, 9, 10] or dynamic profiling of the code [3,4]. In this case, it is necessary to

[*] Informatics Department, Federal Technological Education Center of Rio Grande do Norte, 59015-000 Natal/RN
[‡] Institute of Computing, University of Campinas, 13083-970 Campinas/SP.

intersperse indices and original instructions that are not present in the dictionary to form the resulting compressed code.

Moreover, indices are usually smaller than an instruction word, thus producing a misaligned stream of instructions and indices that frequently requires two accesses to the cache to decompress one instruction. If branch targets are forced to be word-aligned, as usual, extensive padding is used. Finally instructions redundancy may occur when using dictionaries with multiple instructions per entry, because their sequence must be obeyed.

In the present work we propose to pack a set of variable size indices into a fixed size 32-bit word that we call *ComPacket*. This regular size word allows to intermix original instructions with ComPackets eliminating the repeated cache accesses and alignment problems stated 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 form the dictionary, it usually yields better compression results and some benefits on the code performance. On the other hand, performance improvements may be considerably enhanced by choosing instructions based on dynamic profiling information. Thus, we propose a variable mix of static/dynamic instruction count selection criterion to construct one dictionary that performs well for both scenarios.

The best results we obtained point to a reduction in code size of up to 44% while just using static information to build the dictionary, and a reduction in cache accesses of up to 60% when just using dynamic profiling. On average (for our benchmark set excerpted from MiBench and Mediabench suites), code size reduction may be as good as 30% and cache accesses are reduced to 47% of its original.

This report is organized as follow: Section 2 presents the background on code compression and related work. Section 3 explains the dictionary construction algorithm and its trade-offs. Section 4 presents the compression method. The decompressor engine is shown in Section 5. In Section 6 the experimental results are shown and Section 7 presents our conclusions.

2 Background and Related Work

The metric widely used to measure the code compression efficiency is the compression ratio. When using dictionary techniques the dictionary itself becomes part of the compressed object code (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.

$$compression\ ratio = \frac{compressed\ object\ size + dictionary\ size}{original\ size} \qquad \text{(Equation 1)}$$

The key challenge 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) and patching the addresses in the compressed code.

Using ATTs may be prohibitive in PDC architectures because a one-to-one (or at least every basic block label) address translation is required. Patching the code is an operation that has to cope with indirect jumping as well as PC relative branches. Relative branch displacements must be changed to reflect the new position of the targets, and the jump tables need to be patched when using indirect addressing modes. This requires the jump tables' addresses to be the same in the compressed and uncompressed code or the insertion of an extra instruction to fix the register value that points to the jump table. Although time-costly, patching is more suitable to PDC architectures because its overhead happens at compression time,

not affecting run-time performance or the decompressor engine area.

The general outline to perform dictionary based code compression is a 4-step algorithm: building the dictionary; finding dictionary instructions in the original code; encoding the codewords[1]; and replacing original instructions by codewords. An extra fifth step is required when using patching to assign the new addresses to relative branch instructions as well as to fix the jump tables pointers.

2.1 Related Work

Liao *et al* [9] used dictionary code compression for the TMS320C25 Digital Signal Processor. They found sequences of instructions that appear frequently in the code and created *mini-subroutines* in the dictionary (a simple segment of the code). The sequences were then substituted by a *call* instruction, and a *ret* was added at the end of each *mini-subroutine*. This approach has the advantage of using no extra hardware, but the more *mini-subroutine* are used, the greater is the cycle overhead due to the execution of an extra intruction pair *call*/*ret*. In a derived method a new CALD instruction asserts the dictionary entry address and length so that no *ret* instruction is required. Just sequences of two or more instructions are considered candidates to the dictionary, as they yield some compression considering the call dictionary instruction overhead. The compression ratio achieved by the first method was 88% and by the second 84%.

Lefurgy *et al* [10] experiments used fixed and variable-length codewords. The first were used for the PowerPC and the second for the PowerPC, ARM and i386. In the compression method 16-bit fixed-length codewords are used with a 256-entry dictionary, each entry containing 16 bytes, thus allowing 4 instructions in sequence. This occupies 1024 instruction slots. Relative branches are not allowed in the dictionary, facilitating patching. Their approach for 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. This leads to a more complex decompressor to handle all possible situations. Their best results produce compression ratios of 61%, 66% and 74% for the PowerPC, ARM and i386, respectively.

Benini *et al* [4] 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 from the original code. They compress instructions that belong to the dictionary if they can be accommodated in a cache line size package. A mix of compressed and uncompressed instructions may be part of this repository if they yield compression. For every ‘compressed cache line’ the first 32-bit word is used for an escape sequence and a set of flags to indicate the presence of compressed/uncompressed instruction in the remaining 12 bytes of the line. If more than four instructions (up to 12) can be compressed in a cache line then this format is used, otherwise the original cache line (4 words) is kept as it is. A 72% compression ratio is produced for the DLX, but the primary goal of this work was to present the benefits of energy consumption reduction due to dynamic profiling usage. An impressive 35% energy reduction was obtained.

Lekatsas *et al* [5] used the Xtensa 1040 processor to support a compression method based on a small dictionary (256 entries, 1 instruction/entry) based on static instruction count. This processor has irregular instructions sizes of 24 and 16 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 a codeword. 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. Branch targets are word aligned so padding the last word before a target is often necessary. A 65% compression ratio was achieved and a 25% performance improvement (cycles count reduction) was reported.

[1] Codeword is a bit pattern attributed to an index, probably along with an escape sequence to distinguish it from an uncompressed instruction

Our compression method is similar to Benini's in the sense that they pack set of indices, but unlike it, our is completely independent of the cache organization and supports variable size indices. The decompression engine is much simpler when compared to Lekatsas' approach, given that we use a regular size ComPacket that maintain instruction streams word-aligned. We also allow a class of relative branches to be in the dictionary by using a special codification, which contrasts with Lefurgy's approach. Finally, the dictionary construction is based on information obtained by profiling the execution of the code, as well as, the natural choice of static instruction count. This latter feature represents an unexplored design space where we can trade compressibility for performance or energy consumption by changing the dictionary composition.

3 The Dictionary Construction

The composition of a 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 is the best, as we represent with fewer bits the instructions that appear the most. Other researches used dynamic profiling information to guide the dictionary construction. This tends to put into the dictionary the most executed instructions, and may be very effective if some dynamic goal, like bus toggle minimization, is the target. 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 studying is: how different is the static dictionary with respect to the dynamic one? We restrict the answer to the case of small dictionaries with one instruction per entry, as they are more appropriate to the PDC architecture we are investigating. Figure 1 shows that, frequently, less than half of the instructions are common (redundant) to both dictionaries and on average they tend to be less than 30% for a 256-entries dictionary.

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.

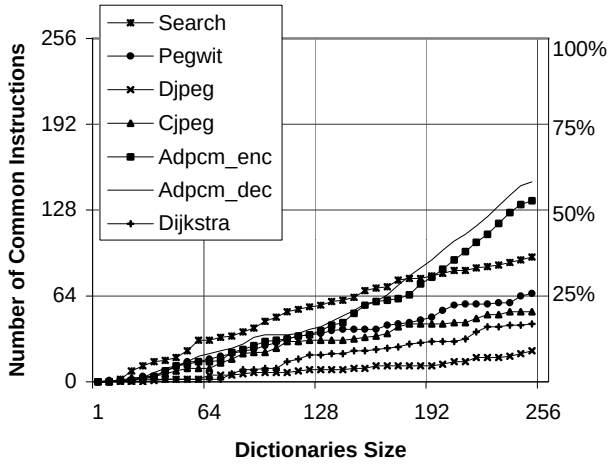

Figure 1: Dynamic X Static Dictionaries Similarity.

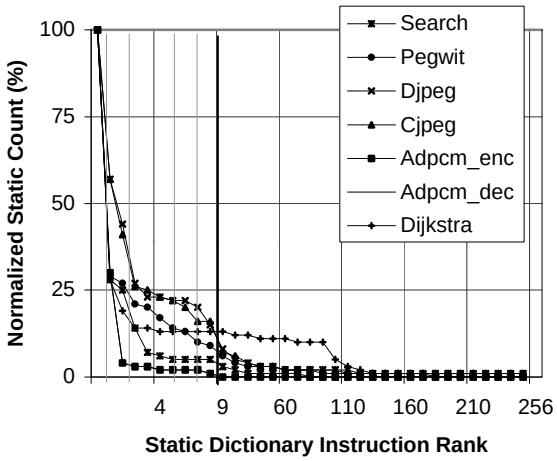

Figure 2: Static Dictionary Instruction count

From these observation we outline a dictionary formation that contains the most static occurred instruction, as well as, the most executed ones. The assumption is that 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.

The algorithm to blend the two dictionaries is a very simple one. We begin by ordering the dictionaries by their natural selection criteria: static count for the static dictionary and dynamic count for the dynamic dictionary. The resulting unified dictionary is called UniD. Then we take the first instruction from SD and check if it is present in UniD. If not, include it. We do the same to include one instruction from DD, and 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 of the algorithm is described in [11].

An advance in this basic algorithm 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 proportion the dynamic factor (*f*) of the UniD. Equation 2 defines the dynamic factor.

$$f = \frac{Instructions\ from\ DD}{Dictionary\ Size} \qquad \text{(Equation 2)}$$

As a result, we can exploit the dictionary formation space to meet special system requirements. Of course, when choosing f = 0%, some instructions from the DD still belong to UniD because there is a small intersection between the SD and DD sets. The dictionary construction algorithm guarantees that at least (f x |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 3 presents the typical shape of the curve we obtain when computing the bus[2] toggle ratio (compressed program bus toggles / original program bus toggles). This graph refers to the Pegwit benchmark. The best compression ratio can be obtained when the dynamic factor is 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, if just 20% of instructions from DD

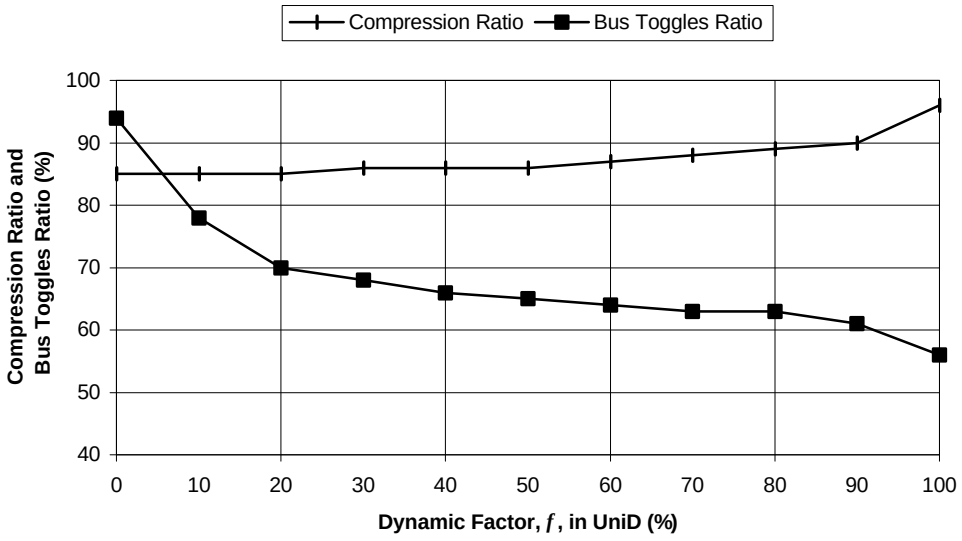

Figure 3: Typical Trade-off Exploitation
(Pegwit benchmark example)

[2] Between cache and decompressor

are allocated to UniD, an expressive reduction in bus toggles is achieved. This reinforces the idea that a small portion of instructions, dynamically or statically chosen, defines the best results.

The conclusion we reach from these experiments is that it is very important to have in the dictionary a mix of instructions from static and dynamic account. It is also clear that choosing pure SD or DD enhances some specific aspect, but modestly improves others.

4 The Compression Method

In our compression method implementation we allow 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). Actually, small branches represent 95% of all branches in our benchmark set and are the majority in typical programs. In order to cope with the offset patching problem in small branches that belong to the dictionary, we keep in the dictionary just the 10 most significant bits of branch instructions 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 not necessarily the first index is a target, thus reducing the required padding from other methods. We currently do not support two or more targets neither two or more small branches in the same ComPacket.

As mentioned before, we use a 32-bit word as a repository for sets of indices. A unique 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 also includes information about the number of indices in the ComPacket, if a small branch offset is present and the target index.

Figure 4 presents the various 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 64 positions are accessible. Format 3 has 3 indices of 8 bits each, thus allowing access to the 256 entries of the dictionary we use. Format 3B has 3 indices of 6 bits each and a branch slot of 6 bits. This restricts the branching offset size, so that just tiny branches (22-bit displacement can be represented with 6 bits) are allowed in this format. Indices are also restricted to point to the first 64 dictionary entries. Format 2B is a repository for two 8-bit indices with possibly one of them being a branch, in the 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.

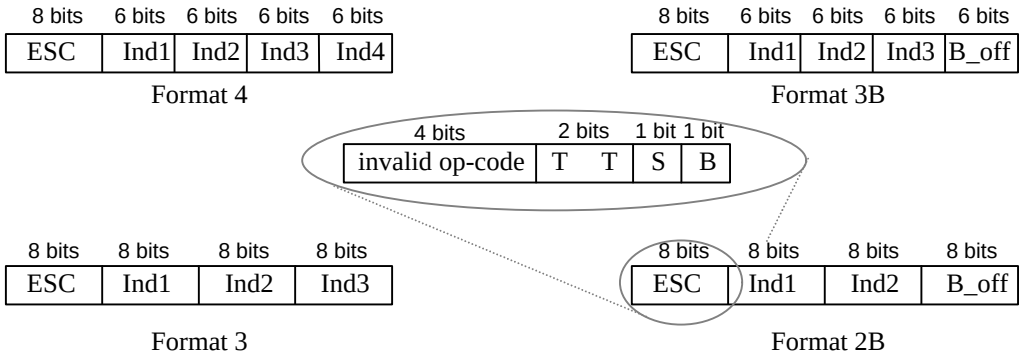

Figure 4: ComPackets Formats

Notice that the branch offsets are 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 does the identification of 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; for TT = 10_2 the third index is a target; and when TT = 11_2 the fourth index is a target. 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 bit fields. They are actually disjointed but uniformly located. Also, the escape sequence is the same for every format.

4.1 The Compression Algorithm

After the dictionary construction step, which was stated in Section 3, we begin the remaining compression algorithm steps, as outlined in Figure 5. In the second step we mark in the code all instructions that belong to the dictionary with a D mark. A T mark is also included for instructions that are targets.

Then, we search in the code all occurrences of the instruction that is the first entry in the dictionary, and whenever we find one we try to form a Format 4 ComPacket. This is possible if its neighborhood has other three instructions that belong to the dictionary and are not marked compressed yet. All of them (now including the instruction under consideration) must point to the 64 first portion of the dictionary; no more than one target should be present; and no branch instruction is allowed in this set. If this all is true, mark these instructions as compressed to form a Format 4 ComPacket. If a Format 4 is not possible, try to form a Format 3 or 3B by analyzing their particular restrictions. Finally, consider a Format 2B. This step is repeated for all instructions in the dictionary, by using a *greedy* approach.

In the next steps we encode the ComPackets marked and begin replacing them in the code. During this step we also create an addresses map from compressed to uncompressed code. Finally, in step 5 we patch the code by using the address mapping appropriately.

We may see an example of the algorithm operation in Figure 6. After the “marking” step (see the D and T marks in figure), we begin by considering the first instruction in the dictionary: *b*. We find in the code the first instance of *b* (at address `04`) and try to form a Format 4 ComPacket. Although *a*, *b* and *c* are dictionary instructions, they cannot form 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`, 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.

```
Compress()
  1. Built the Dictionary (Section 3)
  2. 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
  3. Assembly ComPacket formats marked in 2.
  4. Replace ComPackets in the code
  5. Patch addresses
```

Figure 5: Compress Algorithm Outlined

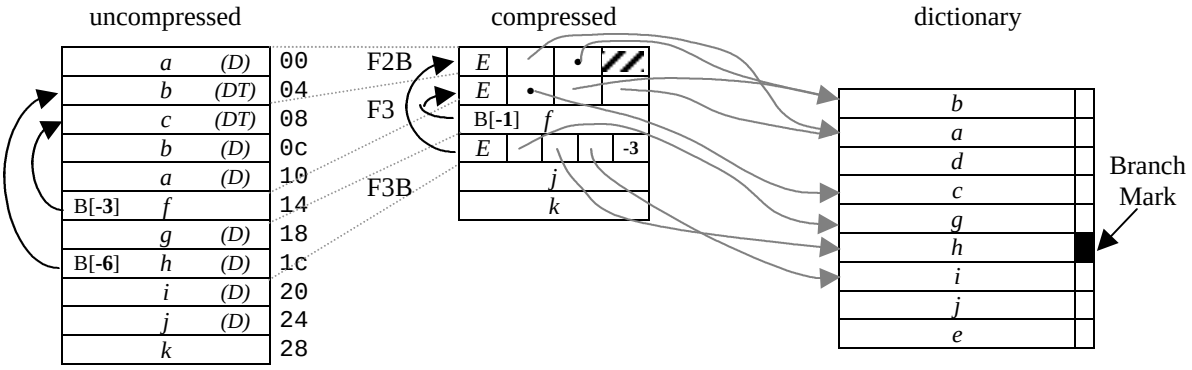

Figure 6: Compression Algorithm Example

As we finish with *b* we consider the second instruction in the dictionary: *a*. No new compression possibility is present. The same happens to *d* and *c*. Now *g* is considered. Although *g* has a neighborhood with three dictionary instructions we cannot pack *g* to *j* into Format 4 because instruction *h* is a branch. To form a Format 3B (*g*, *h* and *i*) we have to check if the displacement of *h* is small enough to be represented with just 6 bits. If so, Format 3B can be formed.

Instructions *h* and *i* in the dictionary do not result in any new compression possibility. When we consider *j* we cannot form any ComPacket because no neighbor instruction is still uncompressed.

The next algorithm step is assembling the ComPackets that were marked in the last step. We can see in dashed lines the ComPackets Formats marked. Now, substitute them into the code. The compressed code is also shown in Figure 6. The gray arrows represent the indices into the dictionary and the dot inside the ComPackets means the target index. The next step in patching the addresses. 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.

5 The Decompressor Engine

The decompressor engine, depicted in Figure 7, performs the following tasks:

- Fetching instructions from cache;
- Finding if the instruction is a ComPacket or an uncompressed;
- If it is a ComPacket, accesses the dictionary and deliver the uncompressed instruction to the microprocessor.

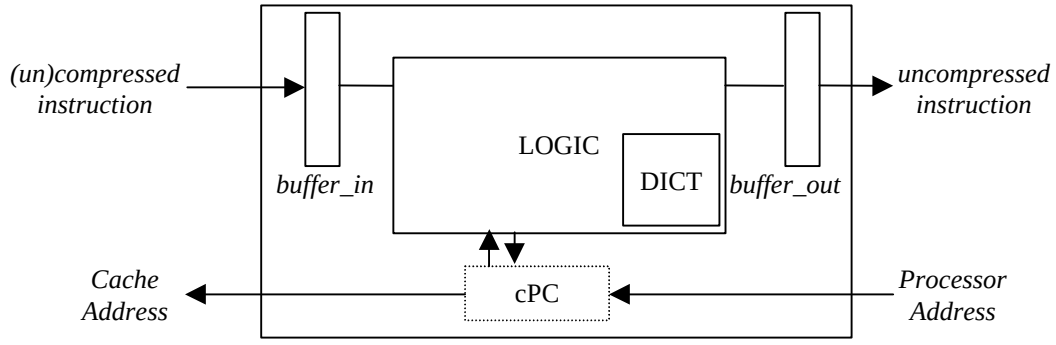

Figure 7: Decompressor Datapath Logic Diagram

The architecture is conceived to perform like a pre-fetch stage of the microprocessor pipeline. It contains two buffers for holding instructions from cache and to the processor; the core logic to perform the tasks listed above; the dictionary itself; and a compressed PC logic (cPC). This latter block is used when the processor core is sealed-off and we cannot change the PC unit to track compressed addresses. In this case cPC has to sense sequential execution or branching to generate the correct address. Notice that the addresses sent to the cache are formed inside the cPC. Thus, when no new 'compressed address' is generated the ComPacket is hold in the *buffer_in* element.

Figure 8 shows the basics of the LOGIC block for several ComPacket formats. The main components are: one MUX to select which index should be used to point the dictionary which is controlled by the cPC unit; one Sign-extensor to be used when a branch offset is inside the ComPacket; and a three-state buffer to allow branches offset to be wired-ORed with the 10 most significant bits of the branch instruction in the dictionary.

To merge each ComPacket logic we augment the MUX index selector for 3 inputs: 2 from cPC and 1 from S. S is also used to select the Sign-extensor for 6 or 8 bits of branch displacement. B bit is used to signal the cPC unit the presence of a branch slot in the last 6 or 8 bits of the ComPacket.

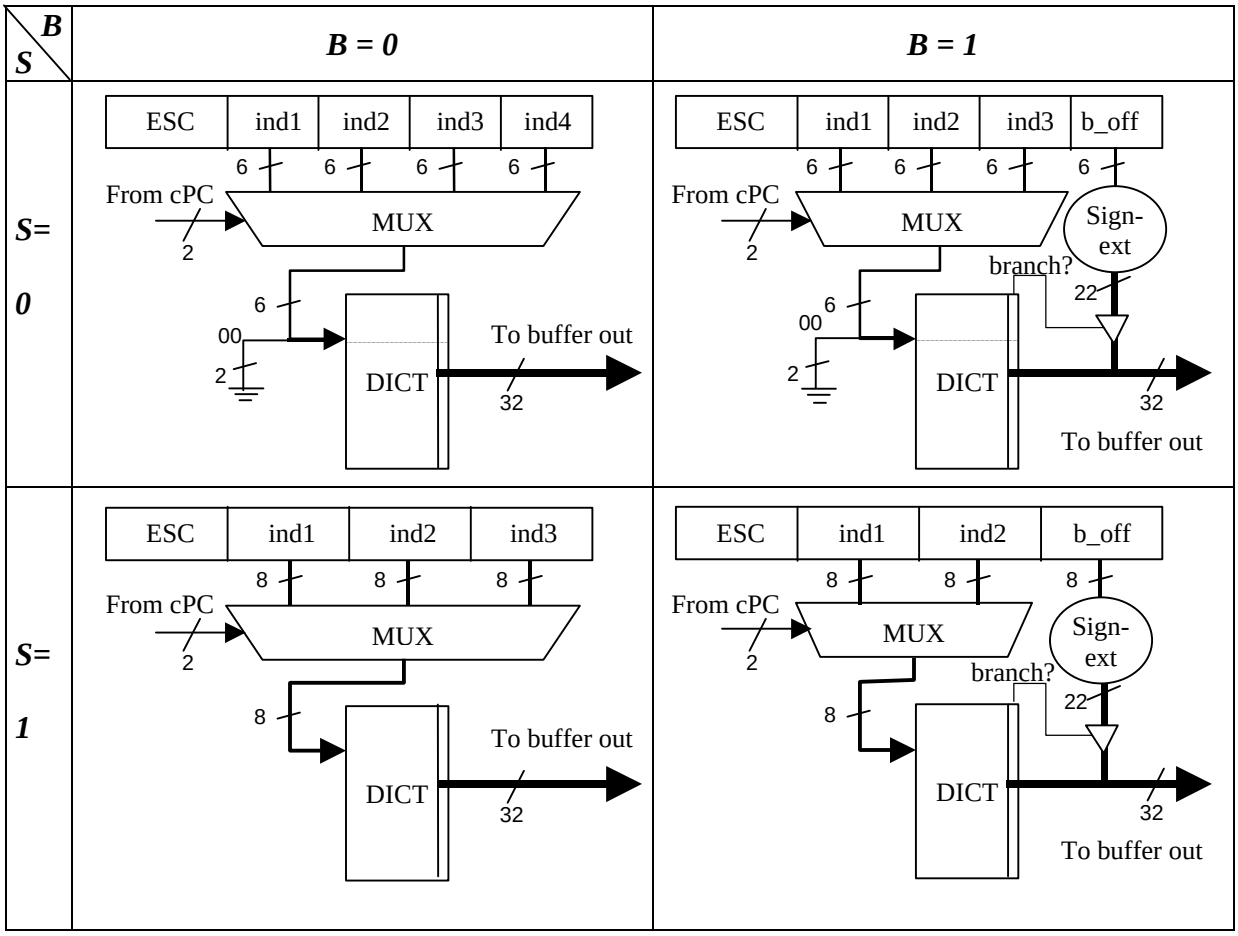

Figure 8: Decompressor LOGIC block for ComPackets Formats in Function of S and B bits

6 Results

The platform for our experiments uses a simulator of the Leon processor (SPARC v8) [12] developed in our lab. This simulator has an interface to the DineroIV cache evaluation tool [13] that we used to measure cache misses.

The benchmarks are extracted from MiBench [14] and Mediabench [15]. They are a string search algorithm, *Search*, commonly used in office suites; *Dijkstra*, an algorithm used in network routers (these two are from MiBench. The remaining we used from Mediabench); *Djpeg* and *Cjpeg*, used for compressing and decompressing images from and to JPEG formats; *Adpcm* encodes or decodes audio; and *Pegwit*, an encryption tool.

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

This Section is organized by the experiments we ran to measure the efficacy of our compression method.

6.1 Compression Ratio vs. Accesses to I-Cache.

We begin by showing the compression ratio results we obtained. Figure 9(a) depicts the complete range of experiments. They exploit all combination 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 this proportion goes up to 90% and compression is severely affected by the dictionary composition.

The next set of results refers to the number of cache accesses that can be avoided. By using a small buffer to hold one ComPacket inside the decompression engine, as we outlined before, we do not need to ask the cache for a next instruction, when it is already present in the buffer. This behaves like a multi-instruction per entry dictionary. Moreover, as the instruction stream is regularly word-aligned and compression does not alter this regularity, we always obtain a complete set of indices.

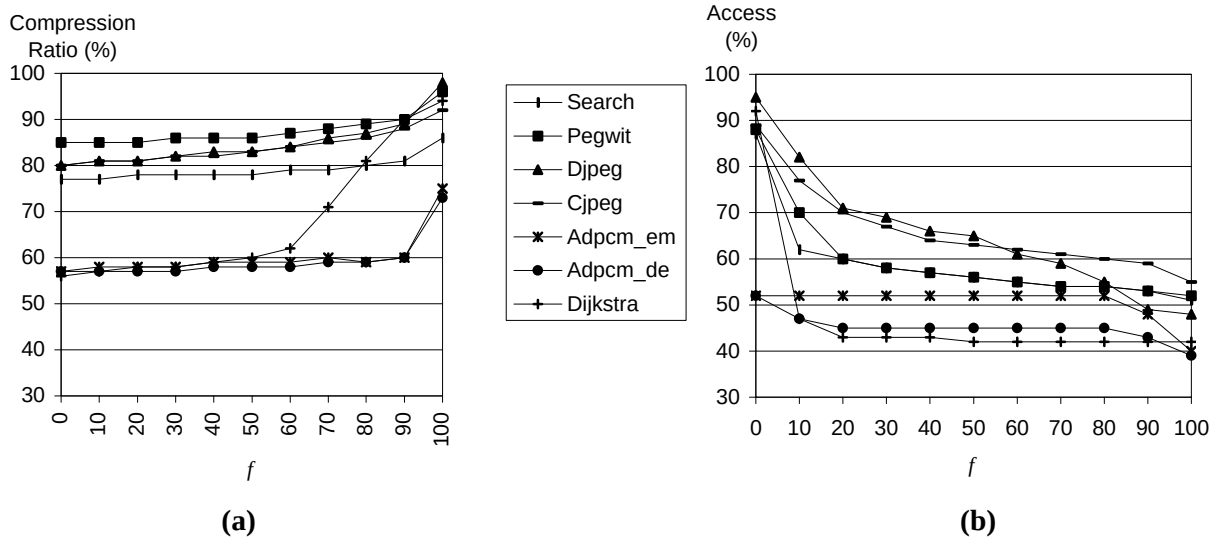

Figure 9: (a) Compression Ratio and (b) Accesses to the I-Cache

Table 1: Bus Toggles

	Original (uncompressed)	f = 0%	f = 20%	f = 50%	f = 100%
Search	112,028,630	101,173,893 (90%)	73,351,773 (65%)	67,899,487 (60%)	60,589,862 (54%)
Pegwit	409,946,579	389,240,744 (94%)	288,630,946 (70%)	267,373,565 (65%)	231,628,100 (56%)
Djpeg	45,878,397	44,939,076 (97%)	35,812,070 (78%)	33,978,211 (74%)	25,014,165 (54%)
Cjpeg	194,398,718	186,658,626 (96%)	152,243,188 (78%)	142,879,988 (73%)	123,775,524 (63%)
Apcm_en	126,567,600	80,139,245 (63%)	74,187,423 (58%)	70,915,803 (56%)	57,605,247 (45%)
Apcm_de	96,667,537	51,858,182 (53%)	46,481,359 (48%)	46,902,895 (48%)	40,603,435 (42%)
Dijkstra	665,188,662	612,586,117 (92%)	375,311,587 (56%)	352,094,985 (52%)	328,979,234 (49%)
	Average	***84%***	***65%***	***61%***	***52%***

Figure 9(b) presents the graphic for every value of f. Notice that cache accesses strongly decrease when f is lower than 20%. After that point an incremental decrease is observed until f=90%, where decrease is again enhanced (although not like the first 20%). This figure can be used to help in the choice of an ideal proportion of instructions in the dictionary to meet specific system requirements.

6.2 Bus Toggles Minimization Between Decompressor and I-cache

We also ran experiments to measure the bus activity between the decompressor and the cache. This represents an important metric to define 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 bit toggles and code bit toggles for four selected values of f: 0%, 20%, 50% and 100%.

These values were chosen because, as mentioned before, when f gets near 20% a significant decrease in bus toggles is detected. 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 we have measured.

On average, we obtained reductions in bus toggles of up to 52%, including address and code. We have also observed that the dictionary formed by 50% of instructions from DD performs less than 10% worse than the best results.

6.3 Cache Behavior

The I-cache behavior was also explored for the entire set of benchmarks. We used a direct mapped cache with line size of 16 bytes (4 words) and varied its capacity from 128bytes to 16Kbytes. These are very small caches but are still representative for the benchmarks size (notice that for the original uncompressed code, miss ratio varies from 10% to 2%, not so different from a real cache behavior). Table 2 resumes the results we obtained for the Pegwit application. For cache sizes of 8Kbytes and 16Kbytes the original miss ratios are just 0.02% and 0.01% respectively, so that we do not present their results in the table.

Surprisingly, we notice that the miss ratio did not reduce by using our code compression approach. In

Table 2: Cache Behavior for the Pegwit Benchmark
miss count / (miss ratio)

Dictionary	Cache Size (in bytes)						Demand Fetches	Dictionary Accesses
	128	256	512	1K	2K	4K		
No-dict Original	3,280,119 (10%)	2,875,657 (9%)	2,249,176 (7%)	2,004,525 (6%)	1,105,670 (3%)	595,089 (2%)	32,976,115	-
f=0%	2,977,959 (10%)	2,528,899 (8%)	2,027,826 (7%)	1,697,666 (6%)	959,175 (3%)	430,156 (1%)	29,150,134	7,495,155
f=20%	2,615,832 (13%)	2,209,324 (11%)	1,925,748 (10%)	1,640,067 (8%)	992,587 (5%)	508,016 (3%)	19,966,634	20,153,916
f=50%	2,400,070 (13%)	2,145,927 (12%)	1,878,292 (10%)	1,647,911 (9%)	1,013,707 (5%)	556,555 (3%)	18,623,917	22,523,737
f=80%	2,309,054 (13%)	2,034,346 (11%)	1,812,218 (10%)	1,633,327 (9%)	1,003,677 (6%)	561,922 (3%)	17,820,968	24,099,546
f=100%	2,283,723 (13%)	2,080,297 (12%)	1,866,110 (11%)	1,607,639 (9%)	914,595 (5%)	539,161 (3%)	17,280,192	17,280,192

fact, we do not access the cache when the processor demands an instruction that is part of one ComPacket already inside the decompression engine. From the cache viewpoint, it is like a completely new workload. So that, forming a dictionary with all instructions from DD does not guarantee cache performance improvement with respect to a static dictionary, specially for larger caches where conflict misses dominates capacity misses and more ComPackets on execution traces mean more conflict misses.

Nevertheless, any compression method tends to reduce the absolute number of misses. We observe that for small caches (<1K in our example) we always find a half sized cache that has the same or lower miss count. This implies that we can exchange the original cache for other of half a size.

We have also studied the number of dictionary accesses during program execution. The more the ComPackets found during execution, the more would be the dictionary accesses. This is supposed to correspond to a modest amount of energy consumption, for example, because dictionary accesses costs are lower than cache accesses, as discussed before.

6.4 A Selected Case for a Dictionary Composition

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 3 summarizes the results. We see that such a dictionary performs very close to the best compression ratio achieved, differing only

Table 3: 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
Pegwit	1	10	85	96	86	56	88	52	32	4
Djpeg	3	15	80	98	83	65	95	48	30	17
Cjpeg	3	9	80	92	83	63	89	55	26	8
Adpcm_en	2	16	57	75	59	52	52	40	0	8
Adpcm_d	1	15	57	73	58	45	52	39	7	6
Dijkstra	4	34	56	94	60	42	92	42	50	0
Average	*2*	*15*							*25*	*7*

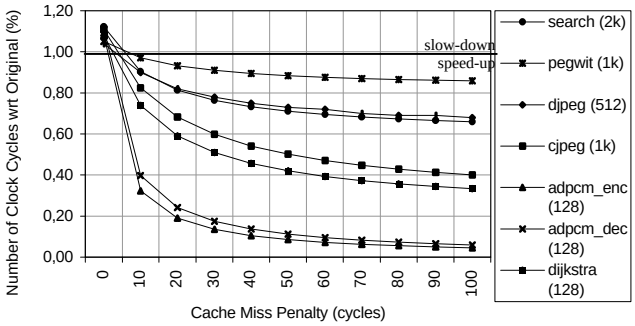

Figure 10: Execution Cycles Relative to Original

2% on average. It also substantially differs from the worst compression ratio in 15%. On the other hand, 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, thus 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 [5] 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 every benchmark a representative cache size, around the point in which an increase in cache size produces just a modest amount of improvement in hit ratio. 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).

In Figure 10 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.

Unfortunately, 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. This implies that the compression ratio could be better. On the other hand, the instruction stream does not have pieces of instructions, as proposed by other researches, considerably improving decompression.

7 Conclusions

So far we have presented a new dictionary based compression method that is independent of the cache and processor and uses a simple low-impact decompressor. The compression symbol used is 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 for architectural parameters that lead to compression, energy reduction and performance at the same time.

The best compression ratio achieved was 56% taking into account the dictionary as part of the

compressed code image. The dictionary represents about 5% of the compressed code in our benchmark set. On average compression ratio is 70%. We also show a reduction of more than 50% in cache accesses.

References

[1] W. Wolf, *Computer as Components: principals of embedded system design*, Morgan Kaufmann, 2000.

[2] A. Wolfe and A. Chainin. Executing compressed programs on an embedded RISC architecture. Proc. of Int'l Symp. on Microarchitecture. pp. 81-91, Dec 1992

[3] Y. Xie, W Wolf and H. Lekatsas. Profile-driven selective code compression. Proc. of DATE'03. pp. 462-467, Mar 2003

[4] L. Benini, A. Macci and A. Nannarelli, Cached-code compression for energy minimization in embedded processor. Proc. of ISPLED'01. pp 322-327, Aug 2001.

[5] H. Lekatsas, J. Henkel and V. Jakkula. Design of one-cycle decompression hardware for performance increase in embedded systems. Proc. of DAC'02. pp. 34-39, Jun 2002.

[6] S. Debray and W. Evans. Profile-guided code compression. Proc. of the ACM SIGPLAN on Programming Language Design and Implementation. pp. 95-105, Jun 2002.

[7] H. Lekatsas, J. Henkel and W. Wolf. Design and simulation of a pipelined decompression architecture for embedded systems. Proc. of ISSS´01. pp. 63-68, Oct. 2001

[8] T. Bell, J. Cleary, and I. Witten, *Text Compression*, Prentice Hall, 1990.

[9] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. Proc. Conf. On Advanced Research in VLSI, 1995.

[10] C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge. Improving code density using compression technique. Proc. of the Int´l Symp. on Microarchitecture, Dec. 1997.

[11] E. Wanderley Netto, R. Azevedo, P. Centoducatte, G. Araujo, Mixed Static/Dynamic Profiling for Dictionary Based Code Compression, To apper in the Proceedings of the Int'l Symp. on System on Chip, Nov. 2003.

[12] G. Gaisler. (2003, May) Leon [Online]. Available: http://www.gaisler.com

[13] M. D. Hill. (2003, May) DineroIV trace-driven simulator. [Online]. Available: http://www.cs.wisc.edu/~markhill/DineroIV.

[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown. MiBench: a free, commercially representative embedded benchmark suite. IEEE 4th annual Workshop on Workload Characterization. Dec 2001.

[15] C. Lee, M. Potkonjak and W. Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communication system. IEEE MICRO-30. pp. 330-337, Dec 1997.