2009 International Symposium on Code Generation and Optimization

Two-Level Dictionary Code Compression: a New Scheme to Improve Instruction Code Density of Embedded Applications

Mikael Collin and Mats Brorsson
KTH School of Information and Communication Technology
Royal Institute of Technology, Stockholm Sweden.
Member of HiPEAC EU Network of Excellence
email: {mikaelco, matsbror}@kth.se

Abstract—**Dictionary code compression is a technique which has been studied as a method to reduce the energy consumed in the instruction fetch path of processors. Instructions or instruction sequences in the code are replaced with short code words. These code words are later used to index a dictionary which contains the original uncompressed instruction or an entire sequence. In this paper, we present a new method which improves on code density compared to previously published dictionary methods. It uses a two-level dictionary design and is capable of handling compression of both individual instructions and code sequences of 2-16 instructions. The two dictionaries are in separate pipeline stages and work together to decompress sequences and instructions. The impact on storage size for the dictionaries is rather small as the sequences in the dictionary are stored as individually compressed instructions, instead of normal instructions. Compared to previous dictionary code compression methods we achieve improved dynamic compression rate, potential for better performance with reasonable static compression rate and with still small dictionary size suitable for context switching.**

Keywords-Dictionary code compression, code density-optimization, code generation.

I. Introduction

Efficient use of chip real-estate and low energy consumption are important design considerations for any computer system, but even more so for embedded systems with their various constraint. Any optimization in code generation, architecture or otherwise that can improve on these factors without compromising performance is worthwhile consideration. One place to look at because of its high activity grade, is the *instruction fetch path* in a processor. It involves fetching instructions from memory – including from the instruction cache or the main memory – performing branch prediction and finally decoding instructions before they can be issued for execution.

Dictionary code compression is the collective name for a class of compression schemes where instructions, or sequences of instructions are replaced with a shorter code word expanded in run-time through a dictionary look-up [2, 4, 11]. Different schemes have been proposed for different purposes. In our case, we are interested in being able to reduce the instruction fetch path energy and to reduce the instruction cache size, without compromising performance or expressability.

Dictionary code compression is based on execution profiles of the applications, which are used to decide the dictionary contents. One way to improve the effectiveness of the scheme is to increase the size of the dictionary. This has important drawbacks such as the need for longer code-words, making them difficult to integrate in the normal instruction set architecture. Another drawback appears if we want to add the dictionary contents to the context of an application in order to better match the dictionary to the program. A larger dictionary then adds significantly to the overhead of context switches. Furthermore, larger code words reduces the effect of the compression.

In our work presented here, we have instead opted for the possibility to increase the *density* of the dictionary contents by means of a two-level approach. The method is based on compression of code sequences using two disjoint dictionaries located in separate pipeline stages. With disjoint dictionaries where the code sequences are constructed from individually compressed instructions, it is possible to fetch up to *16 instructions on a single cache access*. We achieve this without increasing the size of the code words used and with only a moderate increase in total dictionary size.

The focus of this paper is to describe the idea behind the 2-level dictionary compression method, its fundament and the actual method used to achieve compression of sequences using individually compressed instructions. We contribute to the state-of-the-art in the following ways:

- We present an *innovative disjoint dictionary scheme* and the use of *two different code word types*. The result is improved code density with up to 16 instructions in a single fetch word making it possible in many cases to achieve *one basic block — one instruction cache fetch*.
- *Flexible, reusable, and storage efficient representation of sequences. Sequences are represented using code words corresponding to already compressed instructions.*
- *Code transformations for sequence enlargement. We use instruction scheduling in a new way to increase the sequences of compressible instructions.*

We evaluate the proposed 2-level method using simulation of a representative architecture and a set of MediaBench applications [10]. We find that ideally we can reduce the number of bytes fetched from the instruction cache with 80% and 35% with an average of 59%. In a real system with dynamic branch-prediction, these numbers are slightly compromised because of branch mis-predictions, but it typically results in 20% to 50% energy reduction in the instruction fetch path or 15% to 35% of the total processor/memory energy consumption.

The remainder of the paper is organized as follows: In Section II., we present a dense description of some of the related work in the field, which then is followed by an introduction to the proposed 2-level method. In Section IV. we introduce the compression code architecture of our design. The details of the actual compression method is then presented in section V followed by experiments and results in section VI. The paper is then concluded in section VII.

II. Related work

Over the years several researchers have performed studies of *dictionary compression* where instructions or sequences of instructions in the code are replaced by a short code word. Enlisted below are a few approaches with special interest to us and the 2-level approach.

Lefurgy et al., proposed a dictionary compression technique where frequently occurring code sequences are replaced with variable length macro instructions [11]. The primary objective of their method is to reduce the static code size and no emphasis at all is put on dynamic issues. Their approach is based on identifying code sequences up to four instructions in length and place the whole sequence consisting of native instructions in a 16 byte wide dictionary with 256 entries. Depending on the native ISA, they use macro instructions of 8, 12, or 16 bits and achieve a static compression ratio between 0.61 to 0.74 for a set of SPEC CPU INT95 applications. In their paper, they conclude that independently of the ISA used or code word length, the most important factor for high compression ratio is the dictionary size in number of entries. Although with larger macro instructions the static code size increases.

A dictionary code compression scheme that significantly trades off static code size for dynamic code compression is presented by Benini et al. [2]. On average, a dynamic compression ratio of 0.35 is possible that corresponds to 0.40 in fetch path energy ratio. However, this comes with a price of 27% increase in code size. The technique is based on a single dictionary holding the 255 most frequently executed instructions and two disjoint address spaces. The first section contains the compressed parts of the program in form of code words, and the second contains the entire uncompressed original program. The majority of the instructions are fetched from the compressed area. Only upon a miss there will lead to a fetch from the uncompressesd code section.

The presented two-level method has evolved from our one-level dictionary approach, which we also use in our comparisons [4]. The 1-level method is competitive on static compression ratio, although the main purpose is to improve on the dynamic fetch rate to reduce the energy consumption. The solution, however, is based on profile information of individual instructions and therefore not able to optimize the contents of the dictionary when the aim is to compress entire basic blocks.

Hines et al., has presented a compiler-directed code transformation scheme to increase the compressibility of the code [7]. The scheme is presented in the context of their, *Instruction Register File* (IRF) code compression architecture. The technology is based on having a set of dictionaries or Instruction Register Files containing 32 instructions each. The IRFs are used in a register window fashion where an IRF is selected corresponding to the code currently executing. In addition to the *IRF*, the architecture utilizes a 32-entry storage that contains the most frequently used immediate values. In the code up to five instructions can be packed into an *IRF-instruction* which later during decompression is replaced with the original instructions. To enhance the compressibility, increasing the probability to fit as many instructions as possible in the IRF instruction, code transformations are performed. Based on dependence analysis, instructions are re-scheduled both within a basic block as well as between different blocks.

Another technique based on sequences is presented by Lau et al. [9]. Although the primary focus is on static code size, the means to achieve compression is by sequence substitution. The instruction stream is being scanned to identify re-occurring identical code sequences. When a set of identical sequences has been detected, only the first is kept in its original state, the remaining copies of the sequence can then be replaced with an "echo" instruction. This echo instruction act as a reference pointing out the location to the first occurrence of the code sequence. They use register renaming and instruction re-ordering to create larger sequences increasing the number of instructions possible to substitute. They show that the static code size can be reduced with 15-25%, and performance improved by 5-10% depending on the code optimizations used.

Studies of instruction scheduling involving re-ordering of instructions in an effort to minimize the switching activity in between the instructions in a code sequence, has been presented by Parikh,. et al.[12], and by Sinevriotis and Stouratis [13]. Also, Tomiyama, et al. [14], have explored instruction re-ordering with the objective to minimize switching on the bus between off-chip memory and the instruction cache.

Two approaches that significantly can reduce instruction cache energy are *instruction buffering* [1] and the *filter cache* [8]. These techniques are orthogonal to code compression and could both be used in conjunction with code compression.

The method presented in this paper is focused on address access profiling and entire basic block compression using a storage-efficient disjoint two-level dictionary architecture. Compression of sequences is used as a method to increase code density, and instruction re-ordering to enlarge or create longer compressible sequences.

III. Two-level dictionary compression

This section describes the general idea and techniques that distinguishes the two-level dictionary compression to previously proposed dictionary techniques.

A. General idea

In most dictionary code compression schemes the code after compression consists of a mixture of un-compressed instructions and code words that each corresponds to an individual instruction [2, 4, 7]. A different approach is to let the code word correspond to a finite sequence of instructions [11].

In our two-level approach the idea is to combine the two different methods. After compression the code consists of a mixture of uncompressed instructions and code words. There are two types of code words: instruction code words (*ICW*), or sequence code words (*SCW*). The idea is then to extend a traditional dictionary compression architecture with an additional dictionary and sequence de-compression. Although we are inspired by the work of Lefurgy et al.[11], the sequence construction, code word usage, and storage of sequences are rather different to their approach.

A schematic view of the 2-level method, fetch word types and usage of SCWs and ICWs, is shown in figure 1. Whenever the instruction cache is accessed, we assume that a fetch word (*FW*) of 32 bits is fetched. Depending on the compressed code, one of three cases happens. A fetch word can contain up to four sequence words (case **I** in figure 1), up to three instruction code words (case **II**) or one uncompressed instruction (case **III**).

Case I: Here, data fetched from memory corresponds to four SCWs, which each require a complete decompression. One by one the individual SCWs are used to index the sequence dictionary retrieving a sequence word (*SW*) that contains a sequence of instruction code words. In this example the retrieved SW contains four ICWs that each is used to index the instruction dictionary, finally retrieving the original instruction.

Case II: The second compressed format is an ICW only format, which corresponds to 2-3 compressed instructions represented as an instruction code word. Since no sequence word is involved here, we can bypass the sequence dictionary and directly index the instruction dictionary using the ICWs.

Case III: In the third and last case, the fetch word contains only an uncompressed instruction. Since the instruction is not compressed the instruction fetch unit can bypass both dictionaries.

B. Address access profiling

With the possibility to compress sequences of up to 16 instructions in one fetch word, we may be able to fetch one entire basic block in a single instruction cache access. However, the largest basic blocks may not be the largest contributors to the dynamic number of instruction cache accesses as this is also a function of the execution frequency as well as the size of a block.

The two-level compression method is therefore focused on minimizing the number of fetches needed to fetch entire basic blocks, with the goal of: *one basic block, one cache access.* When we have achieved this goal, the number of fetches from a basic block then only depends on the actual number of times the block is executed. It is therefore necessary during the pre-compression profiling to gather information that can be used later during compression to identify which blocks are the most frequently executed. This is done in an off-line functional profile execution, counting the number of times each address is accessed. Later, in the first compression pass, each basic block is assigned a frequency value which in combination with basic block length information is used to decide which block to prioritize for compression and which to leave unprocessed in the first stage. Section VI contains information about basic block sizes and frequencies for the applications used in this study.

C. Properties of two-level compression

In comparison to previous approaches, the 2-level approach improves on code density and usability in the following manner:

A relatively small increase in the total memory space is needed for dictionaries, going from 1 kB to 2.1 kB. Since we use two disjoint dictionaries, we do not need to increase the size of the code words in order to gain access to all entries as in a larger monolithic dictionary. As all sequences in the dictionary are represented by compressed instruction codes instead of full length instructions, the potential over-

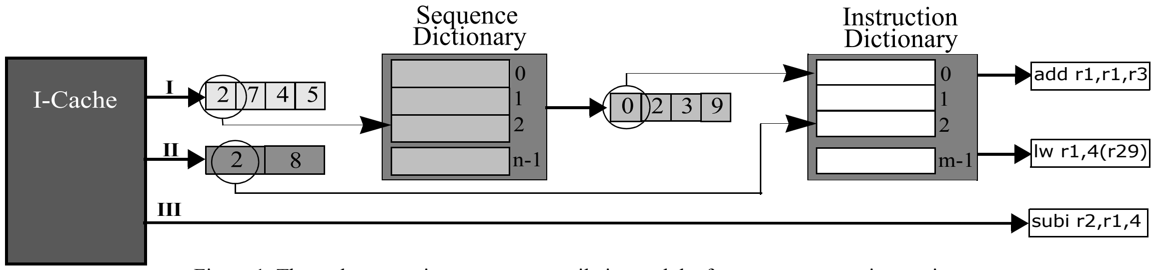


Figure 1. The code generation process, compilation and the four pass compression engine.

TABLE I. FETCH WORD, COMPRESSED CODE CONFIGURATIONS.

cfg-code	Configuration description	CW-dict index	#Inst/ FW	Branch possible	Compressed code layout
0	4 x 7-bit sequence code words	0-127	8-16	Yes	scw scw scw scw
1	3 x 8-bit sequence code words	0-255	6-12	Yes	scw scw scw ▨
2	2 x 8-bit sequence code words	0-255	4-8	Yes	scw scw ▨
3	1 single 8-bit sequence code word	0-255	2-4[a]	Yes	scw ▨
4	3 x 8-bit instruction code words	n/a	3	No	icw icw icw ▨
5	2 x 8-bit instruction code words	n/a	2	No	icw icw ▨
6	2 x 8-bit ICW + one 8 bit disp-value	n/a	2	Yes	icw icw imm ▨
7	Not used				

a. Notice that using configuration 3 to for sequences less than 4 instructions only is efficient if the sequence match the layout of ICW-config 1 or 5 described in Table II.

head of unused space when a sequence entry is not fully utilized is kept rather small: 1-2 bytes.

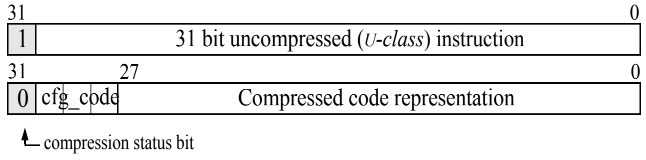
Also, the flexibility is improved due to the design of the sequence code word. As the building blocks of the sequences consist of instruction code words, the same code word can be used to construct several sequences. Also, independently if an ICW is used as a component in a sequence the ICW can always be used to represent an individual instruction.

IV. COMPRESSION CODE ARCHITECTURE

In analogy with an *Instruction Set Architecture* (ISA), we define the *Compression Code Architecture* (CCA) to be the way we encode uncompressed instructions that can co-exist with instruction and sequence code words. It is important that uncompressed instructions may co-exist with compressed instructions as only a small fraction of all the instructions constituting the entire program are possible to encode in a compressed state.

We have extended and modified the MIPS IV ISA binary format to accommodate the use of sequence and instruction code words. Below we define some terms used in the description of the Compression Code Architecture:

- ***Uncompressed instructions:*** *Normal full-length instructions.*
- ***Code words*** *(CW): A generic term for the small values that are used to substitute instructions, or code sequences in the binary. The two-level method uses two different types of code words:*
  1. **Sequence code words** (*SCW*)**:** A SCW corresponds to a compressed instruction sequence. During decompression the scw is used to index the sequence dictionary retrieving a *sequence word* containing *instruction code words*.
  2. **Instruction code words** (*ICW*)**:** In all compressed sequences each individual instruction is represented by an instruction code word. During decompression the ICW is used to index the instruction dictionary where the original uncompressed instruction is stored.


31
0
1
31 bit uncompressed (U-class) instruction
31
27
0
0
cfg_code
Compressed code representation
compression status bit
Figure 2: Fetch word layout.

- ***Sequence word*** *(SW): From an architectural point of view the SW is the 35-bit sequence dictionary entry. In respect to the compression method the contents of each SW corresponds to a sequence of instruction code words.*
- ***Fetch word*** *(FW): A fetch word is the four byte wide entity fetched from the instruction (cache) memory. Each FW either contains a compressed code sequence or an uncompressed instruction.*

A general view of the different CCA components and how they work and produces compressed code can be seen in Figure 1.

A. Fetch word layout

As mentioned before, the fetch word contains either an uncompressed instruction or a sequence of compressed instructions. This is indicated by bit 31 in our Code Compression Architecture, see Figure 2. When the fetch word is a sequence of compressed instructions, they are all from the same basic block, which is an important property from a memory system design point of view.

Defined by the most significant bit, *compression status*, the remaining 31 bits either represent a compressed instruction sequence or one uncompressed instruction. In the case the compression status bit indicates that the data contents correspond to compressed instructions, three additional configuration code bits, *cfg_code*, are allocated from the available data bits to describe the configuration, number and type of code words, used to compress the original instructions as shown in Table I.

Independent of the length of the original code sequences and how many fetch words are required to compress the sequence, the compressed data field of any given FW is confined only to contain code words of the same category. In practice, this implies that the data field either exclusively is composed out of sequence code words (*SCW*), configurations 0-3 in Table I, or only instruction code words (*ICW*), corresponding to configurations 4-7 in Table I. We do not support mixing SCW and ICW within a single FW.

B. Sequence code words (SCW)

Compression of long sequences, 3-16 instructions, are made in a two-phase fashion, where individually compressed instructions are compressed into sequences. A sequence of 3-16 compressed instructions is substituted by one to four sequence code word(s) (SCW) used to index the *sequence dictionary.* As illustrated and described in Table I, sequence code words can be either seven or eight bits long. With seven bit code words, we can only index half of the available entries in a 256-entry large dictionary. For full usage of the dictionary, configuration 1-3 must be used.

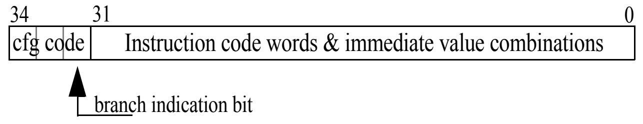


Figure 3: Layout of the 35-bit wide sequence dictionary entry, also referred to as *sequence word(s)* (SW).

The sequence dictionary entry is a 35-bit wide sequence word (SW) with a layout shown in Figure 3. While the fetch word was limited in size to be the same as an uncompressed instruction, we do not have this restriction on the sequence word. Table II shows the layout of the different sequence word configurations. The design of the instruction code words that constitute the sequence words are described below.

C. Instruction code words (ICW)

Instruction code words (*ICW)* constitute the core of the two-level dictionary code compression method. The instruction code word can be defined as a plain 8-bit value used only to index the instruction dictionary located inside the decompression stage in the processor. The resulting dictionary entry is an uncompressed instruction which will be inserted into the processor in place of the code word.

TABLE II. SEQUENCE WORD CONFIGURATIONS.

Code	Code word configuration	Branch	Data layout
0	4 plain instruction code words	No	icw | icw | icw | icw
1	3 instruction code words + one 8 bit immediate displacement value	Yes	icw | icw | icw | imm
2	3 plain instruction code words	No	icw | icw | icw | ▨
3	2 instruction code words + one 8-bit immediate displacement value	Yes	icw | icw | imm | ▨
4	2 plain instruction code words	No	icw | icw | ▨
5	2 instr. code words + one 16-bit immediate displacement value	Yes	icw | icw | imm
6	1 instruction code word + one 16-bit immediate displacement value	Yes	icw | imm | ▨

As described above, the purpose of the 8-bit ICW is to index the dictionary. However, as described later, instruction code words corresponding to branches are used in conjunction with an immediate displacement value. We therefore divide and classify the instruction code words as two different classes, the *G*- and the *R*-class.

G-class: The *Generic*-class corresponds to the definition of the ICW, simply comprising of an 8-bit value possible to substitute for all types of instructions except for relative branches.

R-class: The *Relative-class,* is explicit for compression of relative branch instructions as compression of these instructions is a more complex task. The complexity involved is related to target address generation and the correlation between compressed and uncompressed address space. A low complexity approach to the problem, is to separate the uncompressed displacement value from the compressed op-code and register fields. This simple approach is employed here using two different subclasses of the R-class differentiated by the size and number of bits used in the displacement component.

- **R_8:** The R_8 represents a 16-bit wide compressed encoding of any branch instruction where the branch distance can be expressed using only 8-bits. The 8-bit code used for dictionary lookup is concatenated with an eight bit two's-complement displacement value.
- **R_{16}:** Not all branch distances can be reached using 8-bits. Therefore, to handle longer distances a second R-class ICW using a 16-bit displacement value is introduced.

In the next section we will show how the dictionary contents is constructed based on an address access profile and our fitness-based instruction and sequence word allocation algorithms.

V. COMPRESSION METHOD

A. The Two-level Compression Engine

The compression engine of our technique will in a production system be embedded in the compiler. However, in this study, it is applied as a separate phase on the binary code already compiled. We thus lose the possibility to let our compression engine influence the instruction scheduler which becomes necessary to do as a separate stage now.

The compression engine takes a stream of basic blocks, each containing a stream of instructions. It consists of four distinct passes, *Instruction code word allocation*, *Sequence enlargement*, *Sequence construction*, and finally *Code generation* that step by step transform the instruction stream into compressed code. A small schematic of the process is shown in Figure 4.

B. Instruction code word allocation

The first stage, *Instruction code word allocation*, constitutes the backbone of the proposed method as all compression possible to achieve originates from the use of instruction code words.

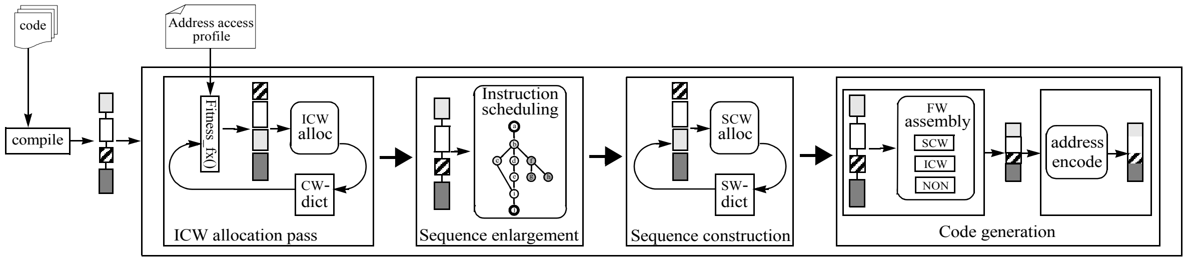


Figure 4. The code generation process, compilation and the four pass compression engine.

From the compiler the compression engine receives a stream of basic blocks that each contains a stream of instructions. Using the address access frequency information gathered during the preceding profiling session, each basic block is assigned an access value corresponding to the number of times each block was accessed during execution. This access information and other properties of the blocks such as instruction count and instruction types are used to calculate a *compression fitness* for the block.

The fitness of a block is then used to decide which block has the best potential to contribute the most to an overall best yield in dynamic compression ratio. The *calculate_fitness()* function is hereby based on a *gain* per *cost* principle, where gain and cost are defined accordingly:

Gain: Gain is a measure on how large effect on the total compression in number of reduced fetches it is possible to achieve if that block were compressed to smallest possible size. With the intention, *one basic block, one fetch word*, a compression of a basic block of nine instructions that are being access 10 times during execution, to fit in a single FW will reduce the number fetches needed for that block from 90 down to 10.

Cost: Compression does not come without a cost. Here, the cost consists of the number if unique ICWs that are required in order to entirely compress the block. Initially the cost to completely compress a block, substitute all its instructions with ICWs, is equal to the number of instructions in the block. However, during the CW-allocation process, the cost is gradually reduced for some blocks as more and more instructions are already present in the dictionary and therefore already have ICWs allocated.

After the initial fitness calculation the stream of basic blocks is sorted in descending order according to their fitness value and the process of allocating entries in the code word dictionary begins. As an instruction becomes allocated in the code word dictionary, all basic blocks that contain that particular instruction must be re-processed. The instruction must be linked to the corresponding code word entry which in turn requires a re-calculation of the fitness for these blocks, as the usage of that code word is now for free. All affected blocks are then re-inserted into the steam of basic blocks in the right fitness order and the process continues until no vacant CW-entries exist. Figure 5 (a) shows the general algorithm of instruction code word assignment.

C. *Properties of the fitness policy*

The adopted fitness policy will promote the compression of a set of small blocks that each individually contributes less than a larger single block contributes to the total, given that the accumulated gain is larger and that the accumulated cost for compressing all the smaller blocks are lower compared to if the larger block was compressed.

Another effect is that blocks that are rather specific in their instruction composition gets a low fitness value, even though they might be frequently accessed. This is because these blocks do not have many instruction patterns in common with other blocks and therefore have a high cost for compression. Blocks with an instruction composition much more similar to other blocks will on the other hand have the

(a)
```
procedure allocate_cw_entries(BB_stream, access_profile)
    calculate_fitness(access_profile)
    sort_BB_stream(fitness)
    while (vacant entries in CW-dict) do
            pick first unprocessed BB
            for each unique instruction pattern
                allocate_cw_entry(pattern)
            end
            for each affected BB
                associate instructions to CW-dict entries
                re_calculate_fitness()
                re_install_BB (fitness)
            end
    end
end
```

(b)
```
procedure allocate_sw_entries(sequence)
len=length_of(sequence)
do
    base=CEIL(len/4)
    nof_icw=(len/base)
    allocate_sw_config(nof_icw, instruction types)
    len=len-nof_icw
while (len>4)
if (len)
    allocate_sw_config(len, instruction types)
end
```

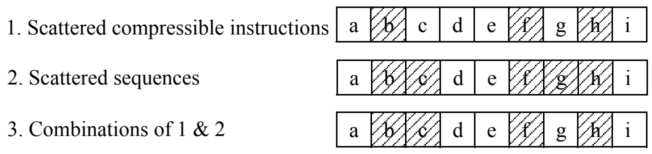
Figure 5: ICW allocation (a) and SCW allocation (b) algorithms.

cost for compression substantially reduced and hence a high fitness value.

D. Sequence enlargement

After the initial ICW allocation phase, the basic blocks are divided into three different sets based on the number of instructions in the block that can be substituted for instruction code words. Set 1 contains the basic blocks where all instructions in the block are in the instruction dictionary. Set 2 contains the basic blocks with two or more instructions that are present in the instruction dictionary and set 3 contain the basic blocks with at most one instruction in the instruction dictionary. This division can be seen in Figure 6. All blocks belonging to set 3 are marked as non-compressible and left unprocessed until the final code generation stage. This is because the block contains too few substitutable instructions. Also, all blocks belonging to set 1 are passed on unprocessed, here because the entire block constitutes a compressible sequence.

In order to achieve any reduction in the number of cache accesses required to fetch the entire basic block, at least two instructions in sequence must be substituted for code words. The instruction code word assignment algorithm, however, does not guarantee that blocks in set 2 have their compressible instructions adjacent. Therefore, we have developed an instruction scheduling algorithm with the purpose of finding compressible instructions so as to create as large sequences of compressible instructions as possible. The process is based on detection and analysis of the three possible cases below.



Dependency analysis. The instruction scheduler depends on a dependence analysis to investigate the possibility to create larger sequences of compressible instructions. Figure 7 shows an example where the objective is to analyze the possibility to merge the two compressible subsequences, {bc} & {fgh}, present in the instruction sequence {a-j} to create a single large compressible sequence {bcfgh}.

The instruction scheduler performs a data dependence analysis on register dependencies only. Although it is possible to some extent detect memory accesses that could precede each other, for simplicity we have chosen to preserve the original load/store order since the improvement in compression ratio because of load/store reordering is insignificant[1]. On average, for the 15 MediaBench applications used in our evaluation, static code size is increased with 0.31%, and the dynamic compression ratio is on average 0.19% lower when strict load/store order scheme is used compared to a test when all possible reordering of load/store-instructions were allowed given that no register interdependences exist between them.

The analysis begins by building a *Data Dependency Graph* (DDG), where the nodes represent individual instructions, see Figure 7. Then, a second DDG is created where each node represents a sequence, uncompressible instructions are here regarded as a single instruction sequence. Any sequence of two or more instructions in length inherit the dependence properties of its contributing instructions as found during generation of the $\text{DDG}_{\text{instructions}}$.

Safe, semantically correct, transformations can now be identified by traversal of the $\text{DDG}_{\text{sequences}}$. During such traversal, a node may be visited only if all the parents of that node already have been visited. Any such valid order of visits induces valid code sequence that do not cause any semantic hazards to the program.

Sequence merge. The merge action is essentially a code transformation, merging the instructions belonging to two yet disjoint sequences, into one larger sequence and to reinsert this new code sequence into the intermediate representation of the instruction stream again on the right location. After a merge action taken place, as long as there are scattered compressible sequence(s), a new DDG reflecting

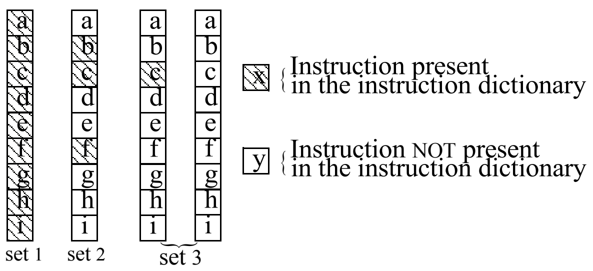

a b c d e f g h i
set 1
set 2
set 3
x { Instruction present in the instruction dictionary
y { Instruction NOT present in the instruction dictionary

Figure 6: Division of basic blocks into 3 different sets based on the number of compressible instruction patterns present.

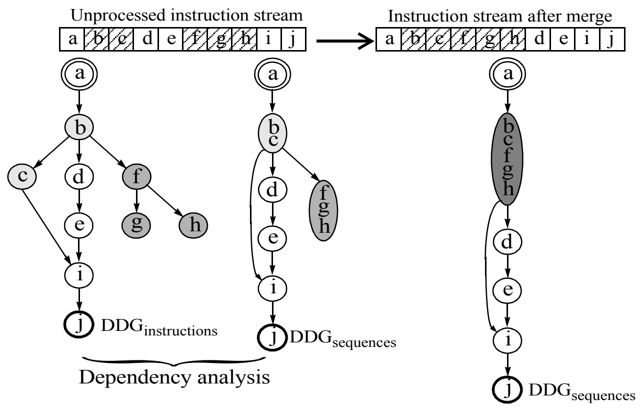


Figure 7: Generation of DDGs to analyse and detect safe, sematically preserving, merge transformations.

1. Despite the low improvement potential on compression rate using a more elaborate load/store order policy, the result is completely unrelated, has no relevance, to other dynamic effects as better memory usage and performance improvements due to dynamic scheduling of load and store instructions.

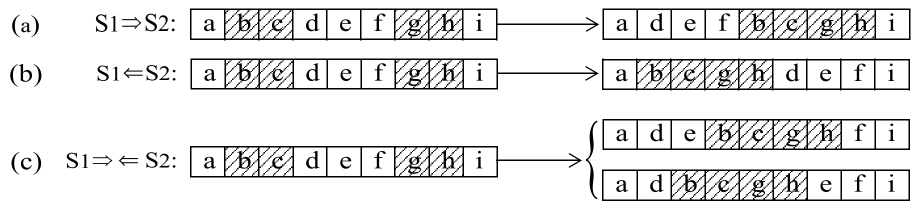


Figure 8: Possible, practical and used merge operations.

the current state of the code is generated and analyzed for more potential merge opportunities.

Previously it has been stated that any valid traversal of the DDG yields a valid permutation of the instruction stream. The actual merge operation restricts this free definition down to three precise transformations described below.

s1 ⇒ s2: Move the fist sequence, s1, forward in the instruction stream to merge together with s2 which is kept in its current location.

s1 ⇐ s2: This transformation is the inverse in respect to previous transformation. Here, the second sequence is moved backward to connect with s1which is kept in place.

s1⇒ ⇐s2: A combination of the two previously described actions. Both sequences are moved towards each other, s1 forward and s2 backward, the two sequences merge somewhere in between.

E. *Sequence word construction*

The responsibility of the SW-construction pass is to identify and process, allocate SW-entries and associate ICWs to the SW-entries, all code sequences possible and adequate[2] to compress using SCWs.

The stream of basic blocks are processed in order of fitness, see Section B., for each block the instruction stream is scanned for one or more sequences to compress. Before SW-allocation can begin the number FWs required for each found sequence must be calculated. The number of FWs required depends on sequence length, available resources in form of empty entries in the SW-dictionary, and whether the last instruction of the sequence is a relative branch or not. The number of FWs is calculated accordingly:

$$\left\lceil \#\text{ICW} + \frac{\#\text{displacement\_bytes}}{\text{max\_len}} \right\rceil$$

where max_len is 16 or 12 depending on whether there are empty sw-entries on index 0-127 or not.

If more than one FW is required to compress the sequence due to a long sequence or resource deficit, it must be subdivided. The sequence is split in two one equal to the current *max_len* in length and a remainder. The remainder is repeatedly subdivided until the length of the remainder ≤ current *max_len*. For each of the FWs needed to compress the entire sequence the sw-allocation procedure is called. See Figure 5 (b) for the sequence word allocation algorithm.

2. Only sequences larger than 3 bytes, ICW and displacement bytes accounted for, are worth compressing using SCWs, smaller sequences are labelled as ICW-compressible and passed on.

The required SWs are allocated in the SW-dictionary in an iterative process. During each iteration the number of ICWs to be assigned to each SW-entry is calculated. The right configuration is selected during the allocation, *allocate_sw_config(),* depending on instruction types and number of ICWs, see Table II.

Index 0-127 are prioritized for sequences larger than 12 bytes as they can be fitted within a fetch word. As long as there are empty entries on index 128-255, sequences with *max_len* less than 13 bytes will be assigned on index 128-255. However, if not enough entries on index 128-255 exist, the 0-127 partition may be used.

All sequences successfully compressed using SCWs are marked as SCW-compressible. If we detect a deficit in SW-entries, the process is aborted and the sequence or subdivided sequence currently processed is marked as ICW-compressible.

F. *Code generation*

Finally the stream of basic blocks reaches the two-pass code generation stage, *Fetch word assembly* and *Address generation*. In the preceding phases, internal compression has been performed building dictionaries and associating instructions to ICWs and ICWs to SCWs. At this point it is time to conclude the process and transform the intermediate representation to a compressed binary file.

Fetch word assembly. Each basic block is once again processed, this time in program order, scanning the code stream locating code sequences belonging to the three compression classes SCW-, ICW-, or non-compressible sequences. Depending on compression class the located sequence is then processed by one of three FW-assembly schemes.

SCW-compression: Since the scw-compression is performed on a fetch word granularity each detected SCW-compressible sequence corresponds to a fetch word. The work here consists of constructing and emitting a fetch word containing the correct SCWs and configuration code to the compressed code stream.

ICW-compression: The sequence to compress is processed from the end towards the start, as long as possible the sequence is subdivided into 3 byte wide units that depending on instruction types can be represented by 2 or 3 ICWs. For each such unit a suitable ICW-configuration is selected from configuration number 4-6 in Table I. If any subdivision of the sequence results in an unit less that 3 bytes in length, those 1 or 2 byte must be handled. For 2-byte units we can employ ICW-configuration 4 to compress the instructions. Any single byte unit is marked as non-compressible and passed on.

NO**-compression:** These instructions are inserted unaltered into the stream of compressed code.

Address generation. After completed compression all branch & jump target addresses must be adjusted to point out the correct destination in the new denser memory space. Depending on branch type and compression state address re-encoding is performed accordingly;

- *For **relative branches** new 8- or 16 bit displacement values must be re-calculated. The update is then either performed on the immediate fields in the sequence word dictionary, or directly on the FW representation of the immediate value.*
- ***Absolute branches** have their target address embedded in the instruction. The adjustment must here either be made on the corresponding instruction dictionary entry or on the actual FW containing the u-class representation of the instruction.*
- *For **indirect branches** which does not hold any address information within the instruction, the corresponding jump tables must be updated with the new target addresses.*

Next we present results from an evaluation of the two-level dictionary instruction compression using 15 programs from the MediaBench benchmark suite.

VI. Experimental evaluation

A. Micro-architecture

We have devised an example micro-architecture that would represent a typical implementation of our scheme, see Figure 9. We assume a small in-order, scalar processor with standard pipeline structure and a dynamic branch-prediction with branch target buffer in the instruction fetch stage. The 2-level architecture is an extension of the more common 1-level approach [4], where a de-pack stage for the dictionary lookup is added. We have added yet another pipeline stage called De-Sequence to perform the sequence dictionary lookup.

In this architecture, the fetch stage does no longer fetch explicit instructions. Instead, a Fetch Word is fetched from the instruction memory. The contents of the FW is inserted into a new *sequence buffer* installed between the fetch- and de-sequence stages. The added buffer act as a FIFO possible to contain the contents of two FWs. A fetch control unit handles the actual fetching and insertion of fetched data. It controls which address is presented to the memory system. The selected address either belongs to the next FW in sequence, or the FW on a predicted target address that is provided by the branch prediction unit (called BTB in Figure 9).

The added DS stage is responsible for reading data out from the sequence buffer and use the data to index the embedded sequence dictionary. The dictionary contains 256 35-bit wide entries. The incoming SCW is then substituted for the ICWs located on the referred SW and inserted into the *code word buffer* between the De-Sequence- and De-Compression stages. In order to put the native instructions into the instruction stream of the pipeline incoming ICWs are used to index the instruction dictionary.

From the Decode stage and onwards, the pipeline remains unaltered as compared to a normal in-order scalar pipelined processor.

B. Methodology.

In order to evaluate our two-level dictionary method, we modified Wattch, a SimpleScalar-based simulator with power models, to model our proposed target architecture [3, 5]. Additional pipeline stage(s), buffers, and dictionaries were added and modelled. To model energy for off-chip memory accesses, the simulator was instrumented with an energy model for memory and bus interface based on the IRAM project [6]. We also compare the new two-level method with our one-level approach [4] as well as a model of a standard five-stage pipelines processor. Simulation parameters for the three different processors can be viewed in Table III.

C. Workload characteristics

We have used the MediaBench benchmark suite to evaluate the effectiveness of our proposed compression scheme

TABLE III. PROCESSOR SIMULATION PARAMETERS.

Processor:	
Misspredict penalty	Baseline: ***3***, 1-level: ***4***, 2-level: ***5*** cycles
2-bit bimodal predictor	1024 entries
Branch target buffer (BTB)	128 entries, direct mapped
Return stack	8 entries
Memory system	
L1 I-cache	16kB, 4-way, 32 B blocks, 1 cycle latency
L1 D-cache	16kB, 4-way, 32 Bblocks, 1 cycle latency
TLB (D&I)	128 entry, 4-way, 30 cycle miss penalty
Main memory	64 cycle latency
Energy and process parameters	
Feature size	0.18 µm
Vdd	1.8 V
Clock frequency	400 MHz

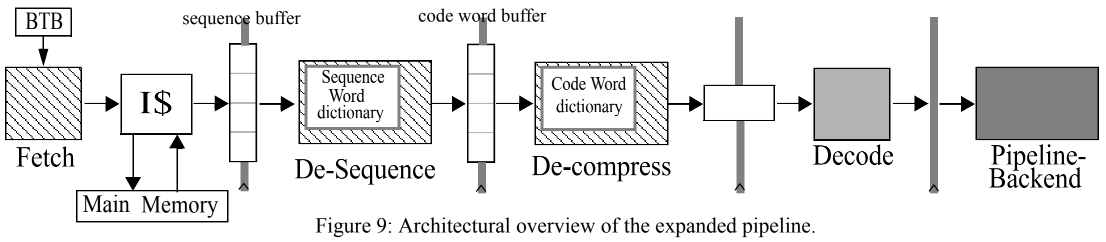


Figure 9: Architectural overview of the expanded pipeline.

	ADPCM		G721		JPEG		MESA			PEGWIT			EPIC	GSM		Mean
	rawcaudio	rawdaudio	encode	decode	cjpeg	djpeg	mipmap	osdemo	texgen	generate	encrypt	decrypt	epic	toast	untoast	
Number of BBs	2383	2381	2813	2812	6399	7193	22435	23453	22454	4050	4050	4050	4452	4282	4282	7833
AVG BB_length (insts)	4.12	4.13	4.32	4.33	4.94	4.99	5.66	5.69	5.65	5.50	5.48	5.48	4.41	4.87	4.87	4.962
Static compression ratio	0.86	0.85	0.86	0.86	0.86	0.91	0.93	0.94	0.93	0.89	0.89	0.89	0.89	0.94	0.88	0.89
Ideal dynamic comression ratio	0.33	0.34	0.35	0.35	0.37	0.44	0.41	0.25	0.49	0.21	0.40	0.43	0.20	0.65	0.27	0.37
# BB fully compressible	118	123	99	95	97	59	105	100	116	111	138	115	178	36	131	108.1
AVG inst/all fully compressible blocks	4.73	4.88	6.04	5.99	6.07	6.29	5.63	5.08	4.98	4.87	4.15	4.69	3.76	9.56	4.57	5.42
# BB fully compressible using SW & CW	79	83	59	59	68	39	58	52	55	62	63	58	98	27	67	61.8
AVG inst/sequence compressed	6.19	6.34	9.02	8.66	7.79	8.56	8.05	7.54	7.55	7.16	7.11	7.53	5.52	12.41	7.13	7.771
# BB fully compressible using only CW	39	40	40	36	29	20	47	48	61	49	75	57	80	9	64	46.27
AVG inst/sequence compressed	1.77	1.85	1.65	1.61	2.03	1.85	2.64	2.42	2.67	1.98	1.67	1.79	1.61	1.00	1.89	1.895

Figure 10: Main application characteristics of the 15 MediaBench applications.

[10]. They were compiled without modification with gcc version 3.2.2 with optimization level -O2.

Figure 10 shows the main characteristics of the 15 programs from MediaBench and static results from applying the two-level compression scheme. The average basic block size in these programs are between 4 and 6. The diversity in basic block size, however, is high. In particular, toast has a dominant basic block of 401 instructions which effectively blocks compression. The average static compression ratio is 0.89 to be compared with the 0.73 achieved by the one-level approach [4]. The difference comes from the fact that the 1-level method can make use of scattered single compressible instructions while in the 2-level approach we need at least a sequence of two compressible instructions. All programs except for djpeg and toast which have around 100 basic blocks which are fully compressible of average length 4-10 instructions. On average 60% of the fully compressible blocks used sequence words, the remaining only instruction code words. The basic blocks compressible using sequence words have a much larger average number of instructions with a mean of 7.7.

D. Dynamic Compression

Although static compression may be important in some embedded applications, it is becoming less so with compact and dense instruction memories. More important is the working set size and the dynamic compression ratio which directly translates into energy savings.

Figure 11 shows the dynamic compression ratio of our two-level approach compared to the one-level approach [4]. The graph shows both the ideal compression ratio, and the real dynamic compression ratio taken from executions on the simulator where the dynamic effects of the branch predictor becomes visible. For programs with relatively high branch prediction miss rate such as rawcaudio and rawdaudio, this is visible as a relatively large difference between the ideal and the real cases. These programs have 20-27% branches and a branch prediction accuracy as low as 70-85%.

The two-level compression scheme is consistently better performing than the one-level approach due to the emphasis on basic blocks and its ability to compress longer sequences into a single fetch word. The exception is toast where a single commonly executed basic block of 401 instructions blocks the compression because it would require more instruction code words than what we have available.

E. Front-end energy consumption

Figure 12 shows the processor front end energy ratio broken down into its components for the 1- and 2-level architectures, normalized to the front end energy of our baseline processor. The energy consumption in the fetch-path is broken down into energy in the decompression stages (mainly dictionary lookup), Branch predictor (denoted BTB), Instruction memory and bus and the Instruction cache. Of these four components, by far the most important is the Instruction cache. The solid line on top of

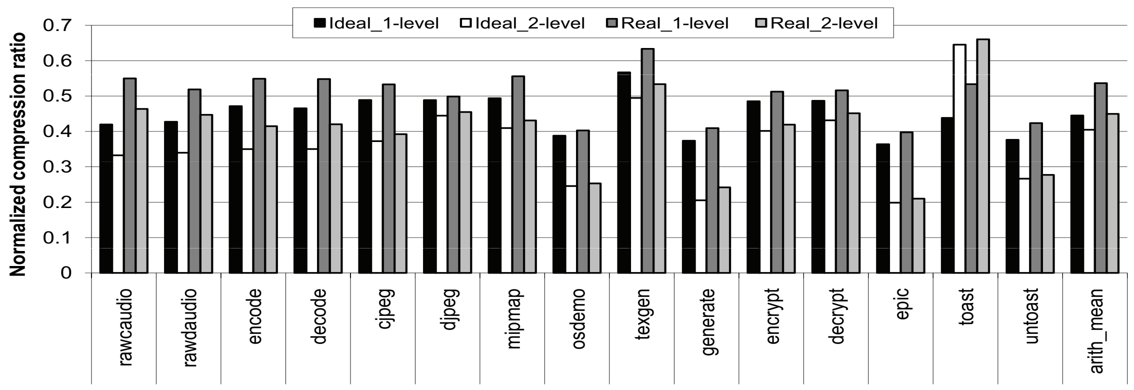


Figure 11: Dynamic compression ratio.

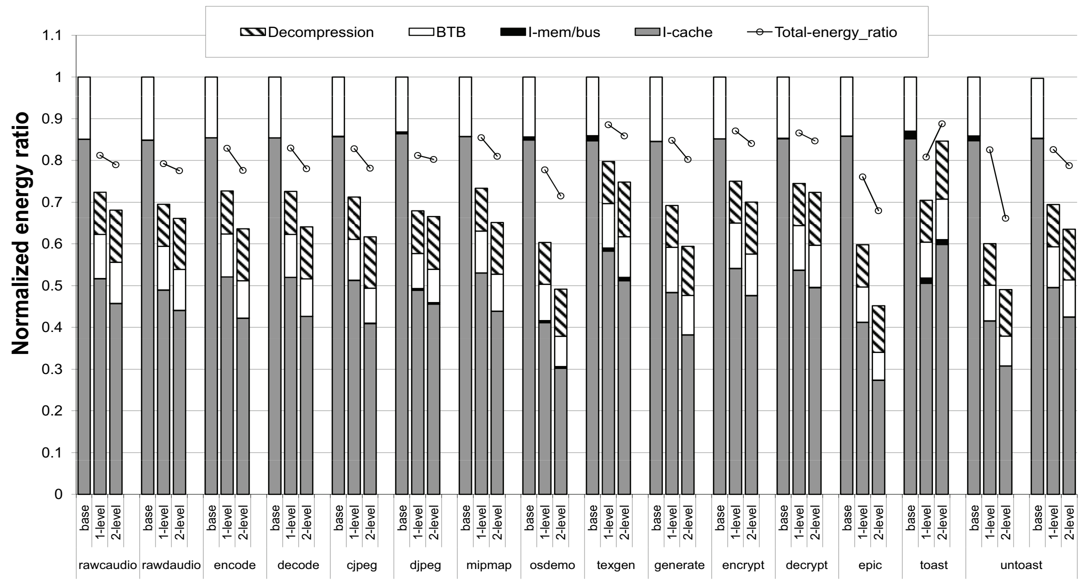


Figure 12: Energy ratios of the Instruction fetch path and total processor/memory.

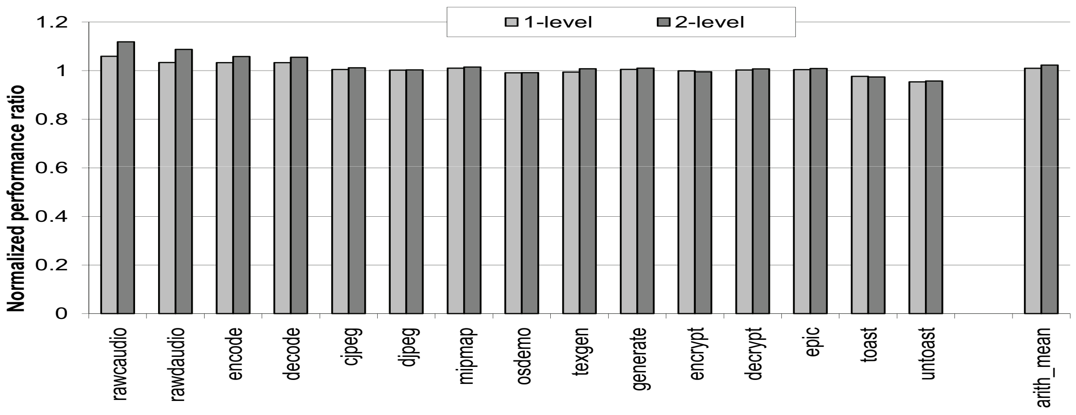


Figure 13: Performance ratio, normalized to the baseline cpu.

the one-level and two-level bars show the reduction in total processor-memory energy from using the two different dictionary code compression schemes.

For the programs in this workload and the instruction cache size used in the experiments, the instruction cache misses are insignificant and therefore very little energy consumption in the instruction memory and bus system. From Figure 12, it is clear that the results regarding dynamic compression ratio shown in Figure 11 directly reflects how the energy consumption in the instruction cache is affected, the better compression ratio the lower I-cache energy consumption. Also, the energy consumed on BTB lookups are related to the dynamic compression, fewer I-cache accesses lead to fewer BTB lookups and vice versa.

Even though the decompression mechanism consumes a considerable amount of energy, for all the test programs used except for toast, the two-level architecture proves to improve on front-end and total energy consumption as compared to our baseline processor.

In order for the 2-level architecture to improve on front end energy over the 1-level architecture the reduction in instruction cache access and BTB lookups must translate into a larger energy reduction than the extra energy consumed by the new de-sequence stage and dictionary adds to the total[3].

F. *Performance*

The immediate relationship between, dynamic compression, and reduced number of instruction cache accesses potentially resulting in improved energy consumption in the front end, does not necessarily apply to performance. Figure 13 shows the performance of the two dictionary approaches normalized to the baseline processor.

Most programs perform equal to the baseline processor. Rawc/Rawdaudio have, as mentioned previously, poor branch prediction accuracy and this shows in lower performance using compression since the branch miss prediction

3. That is, given that the number of instruction cache misses are rather equal for both architectures.

penalty is higher. Some programs, such as encode/decode exhibit dramatic performance (and energy) improvements with very small caches since the compressed instruction stream has a much smaller working set and thus can fit better in the cache compared to the baseline processor. Experiments with a 4 kB instruction cache show, for these programs, a performance ratio of 0.45 and 0.5, respectively, i.e., a reduction of the execution time by a factor of 2.

VII. CONCLUSIONS

In this paper, we have introduced and presented a two-level dictionary code compression method. Based on a method that could be viewed as a traditional dictionary code compression architecture using a single dictionary for decompression, the two-level is an extension using two separate dictionaries, one for compressed instructions not particularly different from what normally is the case for dictionary compression. The new, second dictionary on the other hand contains compressed code sequences. The novelty of the approach is in fact the use of two separate dictionaries and that the compressed sequences are in fact built up by individually compressed instructions. In addition to the presented method the means for compression and decompression are presented, code word architecture, micro architecture and a compression engine capable of utilizing the proposed method.

The efficiency of the new two-level method and architecture has been evaluated on four measurements static code size, dynamic compression, front end energy consumption, and performance impact, against a more traditional dictionary code compression architecture here denoted one-level.

On static code size, the two-level runs short compared to the one-level method. It is an active choice to let the profiling method and compression method trade off some of the performance in static code size for better results regarding dynamic compression.

This is also reflected when looking at the dynamic compression, i.e., the reduced number of cache accesses. It is encouraging to note that with an average improvement of 23% on dynamic compression, between 2-21% reduction in front-end energy consumption is achieved in spite of the now larger and more energy-consuming hardware required for the actual decompression, comprising of two pipeline stages and two dictionary tables.

The introduced extra de-sequence stage does not only effect the results because its own energy consumption, it also increases the mispredict penalty. This makes the architecture both from a performance and dynamic compression point of view extra sensitive to the misprediction rate especially for application executing a large share of branch/jump instructions. For applications where the prediction accuracy is good the impact on performance is within 0.2 -6% compared to the one-level processor.

Currently the proposed compression method is un-optimized and has a considerable high complexity repeatedly throughout the process scanning and sorting the entire stream of basic blocks. In future implementations more effort must be made to reduce the complexity that also involves to make the process integrated in the compiler in contrast to now being a post-compile procedure.

REFERENCES

[1] R. S. Bajwa et.al, *Instruction Buffering to Reduce power in Processors for Signal Processing*, IEEE Transactions on Very Large Scale Integration, Volume 5, No 4 December 1997, pp 417-424.

[2] L. Benini, F. Menichelli, and M Olivieri, A *class of code compression schemes for reducing power consumption in embedded microprocessor systems*, IEEE Transactions on Computers, Volume 53, Issue 4, April 2004 Page(s):467 - 482

[3] D. Brooks, V. Tiwari V., and M. Martonosi, Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000, pp. 83-94.

[4] M. Brorsson and M. Collin, Adaptive and Flexible Dictionary Code Compression for Embedded Applications, in *Proceedings of the ACM/IEEE international conference on compilers, architecture, and synthesis for embedded systems*, October 2006.

[5] D. Burger and T. M. Austin, *The SimpleScalar Tool Set, Version 2.0*, Computer Architecture News, June 1997, pp. 13-25.

[6] R. Fromm et al., The Energy Efficiency of IRAM Architectures, in *Proceedings of the 24th Annual International Symposium on Computer Architecture, ISCA'97*, Denver, CO, 2-4 June 1997, pp. 327-337.

[7] S. Hines, D. Whalley, and G. Tyson, Adaptation Compilation Techniques to Enhance the Packing of instructions into registers, in *Proceedings of the ACM/IEEE international conference on compilers, architecture, and synthesis for embedded systems*, October 2006.

[8] J. Kin, M. Gupta, & W. H. Mangione-Smith, The Filter Cache: An Energy Efficient Memory Structure, in *Proceedings of the 30th International Symposium on Microarchitecture*, Dec 1997, pp. 184-193.

[9] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder, Reducing Code Size With Echo Instructions, in *Proceedings of the ACM/IEEE international conference on compilers, architecture, and synthesis for embedded systems*, November 2003.

[10] C. Lee, et al., MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems, in *Proceedings of the 30th International Symposium on Microarchitecture*, Dec 2997, pp. 330-335.

[11] C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge, Improving Code Density Using Compression Techniques, in *Proceedings of the 30th Annual International Symposium on Microarchitecture, MICRO'30*, December 1997, pp. 194-203.

[12] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, *Instruction scheduling based on energy and performance constraints*, in Proceedings of IEEE Computer Society Workshop on VLSI, 2000. 27-28 April 2000 Page(s):37 - 42

[13] G. Sinevriotis and T. Stouraitis, *A novel list-scheduling algorithm for the low-energy program execution*, IEEE International Symposium on Circuits and Systems, 2002. ISCAS 2002. Volume 4, 26-29 May 2002 Page(s):IV-97 - IV-100 vol.4

[14] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura, *Instruction Sceduling for Power Reduction in Processor-Based System Desig*n, In Proceedings of Design, Automation and Test in Europe, 1998.23-26 Feb. 1998 Page(s):855 - 860