A Performance and Energy Exploration of Dictionary Code Compression Architectures

Mikael Collin, Mats Brorsson
Dept of Software and Computer Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: mikaelco@kth.se, matsbror@kth.se

Johnny Öberg
Dept of Electronic Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: johnnyob@kth.se

***Abstract*——We have implemented and evaluated a novel dictionary code compression mechanism where frequently executed individual instructions and/or sequences are replaced in memory with short code words. The result is a dramatically reduced instruction memory access frequency leading to a performance improvement for small instruction cache sizes and to significantly reduced energy consumption in the instruction fetch path.**

We have evaluated the performance and energy implications of three architectural parameters: branch prediction accuracy, instruction cache size and organization.

***Index Terms*—Code compression, low energy architecture, CPU architecture;**

I. Introduction

Dictionary code compression is the collective name for a class of compression schemes where instructions, or sequences of instructions, are replaced with shorter code words and expanded in the core during run-time through a dictionary look-up [?], [?], [?]. Different schemes have been proposed for different purposes and 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 functionality in the usage of the instruction set.

With this in mind we have previously presented and evaluated a proposal for a two-level dictionary code compression scheme [?], [?] resulting in 20-50% reduced energy consumption in the instruction fetch path. In this paper, we study different design alternatives in more detail in order to understand the trade-offs better. Specifically we contribute with the following:

- An in-depth *micro-architectural description* and analysis of our novel 2-level dictionary code compression method including an implementation of the critical parts using standard cell synthesis technology.
- An extensive evaluation of architectural design alternatives such as *dictionary sizes, cache organization, pipeline alternatives* and *branch prediction performance* and the resulting effect on performance and energy consumption.
- We have also considered combinations with complementary techniques such as *filter caches*.

The net result is a dramatically reduced number of accesses to instruction memory, effectively fetching more instructions at every access which leads to reduced energy consumption in the instruction fetch path as well as the illusion of a virtually larger instruction cache. The latter can be used to reduce the instruction cache while keeping the same performance and therefore further improving on energy consumption, cache hit latency and silicon area. The technology is most useful for embedded systems where energy (or power) consumption is of high priority but also in multicore system where the aggregate cache is large but the available cache for each core is small.

II. Dictionary Code Compression and other Related Work

Most dictionary code compression schemes stores the compressed code as a mixture of uncompressed instructions and code words that corresponds to an individual instruction [?], [?], [?]. Lefurgy et al., presented another approach where whole sequences of instructions were replaced with a single code word [?]. Our two-level approach is a combination of these two different techniques. The compressed code still contains code words and uncompressed instructions, the difference is that we here employ two types of code words: instruction code words, and sequence code words. The fetch stage retrieves a *fetch word* of four bytes from the instruction cache. A fetch word can contain up to four sequence code words, which each can represent up to four instruction code words. A fetch word can thus, in the best case, represent a maximum of 16 uncompressed instructions in one fetch.

The code words are selected based on profile measurements on the frequencies of instructions and can be done in several ways. The 2-level scheme and the compression mechanism, including code word selection, are extensively described in earlier publications [?], [?] and out of the scope of this paper which focuses on the micro-architecture in more detail and the trade-offs we face.

A complementary methodology to reduce the energy consumption is to use a filter cache as described by Kin et al. [?]. The idea is that an extremely small cache will filter out most of the energy hungry references to the rest of the memory system. Preliminary results show that the effects of the filter cache and the proposed dictionary code compression are synergistic and we can reap the benefits of both approaches.

Several other dictionary code compression techniques have been proposed before, e.g. [?], [?], [?], [?]. Our approach differs in that we investigate a wider range of architectural parameters and also consider modern high-performance design alternatives such as the branch predictor used for speculative execution.

We will now proceed and describe the micro-architecture and its design alternatives of our proposed design in more detail.

III. Micro-architecture

Figure 1 shows an overview of the architectural additions needed for the decompression mechanism studied here. The instructions stored in memory contain a mix of code words and uncompressed instructions. We have devised a method to be able to distinguish code words from uncompressed instructions looking at one bit only as described in [?], [?].

The instruction fetch stage fetches a word (FW: four bytes) at a time from the instruction cache based on program counter value or the branch prediction mechanism. The fetch word is then inserted into the fetch buffer located between the Fetch- and Desequence stages. There is also a corresponding code word buffer between the Desequence and the Decompress stages. The Fetch stage then stalls as long as new instructions can be decompressed by the combination of Desequence and Decompress and this is where most of the energy benefits of our technique comes from.

A. Sequence and Instruction Decompression

The task of the Desequence stage is to process the fetch word provided by the fetch stage and to deliver an uncompressed instruction or a sequence of 2-4 instruction code words to the Decompression stage as displayed in figure 2 (a). In addition to the sequence dictionary, which is the core of the Desequence stage, we also find the sequence buffer and various control logic.

The fetch buffer contains one fetch word and is partitioned into a *config, head* and *tail* segments. The config segment determines which of the possible seven code word encodings (see [?], [?] for more information on this) will be used. If the code word represents an uncompressed instruction, or only contains instruction code words, the whole fetch buffer is just passed on to the Decompression stage.

For compressed sequences the head is forwarded to mux2, and the tail is passed through mux1 to the barrel-shifter and inserted into the sequence buffer. The semantics of the two segments are accordingly:

- Head: bit position 27-20, corresponds to the first byte of any fetch word that contains sequence code words. The head is, given that the sequence buffer is empty, passed to the sequence dictionary where 7 or 8 bits are used to index the dictionary.
- Tail: this includes the head for the cases when the sequence buffer is not empty. For this reason, the barrel-shifter shifts the tail to remove the head, if it already has been passed to the sequence dictionary.

The sequence buffer is the remainder of the fetch buffer when the config segment has been read and the head used to access the sequence dictionary. In parallel with the head of the fetch buffer is passed to the dictionary, the shifted tail is inserted into the sequence buffer left shifted 7 or 8 bits depending on the size of the sequence code word. The sequence buffer can also be regarded as partitioned in two segments, *next* and *end*, that have the same function as the head and tail segments of the fetch buffer. As long as the end segment contains a valid sequence code word, the control logic will each cycle control mux2 to pass data from the next partition through to the dictionary, and mux1 to pass data from the end partition through the barrel-shifter.

The sequence dictionary is a memory array with 256 35-bit wide entries. The code word dictionary, used in the next stage, has also 256 entries but here the entries are determined by the instruction format and thus bound to be 32 bits. We will later investigate the effect of differently sized dictionaries. Both dictionaries can either be implemented as a ROM or RAM cell depending on whether the contents are to be fixed at fabrication time or possible to change during run-time through software mechanisms [?]. Although it is necessary with a fixed entry-size of 35 bits to contain all data and configuration bits, our analysis indicates that 64 sequence dictionary entries are sufficient and at the same time more effective and power efficient that utilizing the whole index capacity of 256 entries.

The objective of the Decompression stage is to expand the instruction code words from the previous stage into native instructions using dictionary look-up or bypass uncompressed instructions into the instruction register, see figure 2 (b). After the Desequence stage has performed its task, the code word buffer either contains a single uncompressed instruction or 2-4 instruction code words. The config-bits determines the interpretation of the code word buffer and the decompression schemes mux used in constructing the final instruction stream to the pipeline back-end.

Figure 3 shows the four different actions done in the decompression stage. The first byte, here labeled with subscript 0, is always used to index the dictionary. The dictionary responds with the 32 bit-pattern that corresponds to the native instruction the code word was substituted for during compression. Four different schemes are used for the decompression depending on the code word type. Type G stands for a simple one byte code word representing a single instruction. Type R8 or R16 stands for a type where the code word is supplemented with an immediate value (8 or 16 bits) appended to the result from the dictionary. Finally, type U stands for an uncompressed instruction and the dictionary is not consulted.

B. Architectural design parameters

There are several aspects to consider even if the baseline decompression design as outlined above. The size of the dictionaries is just one of them. Since compression affects the number of effective instructions stored in the cache line, the cache organization, size and line size are also of interest. Although we have presented a design where desequencing an

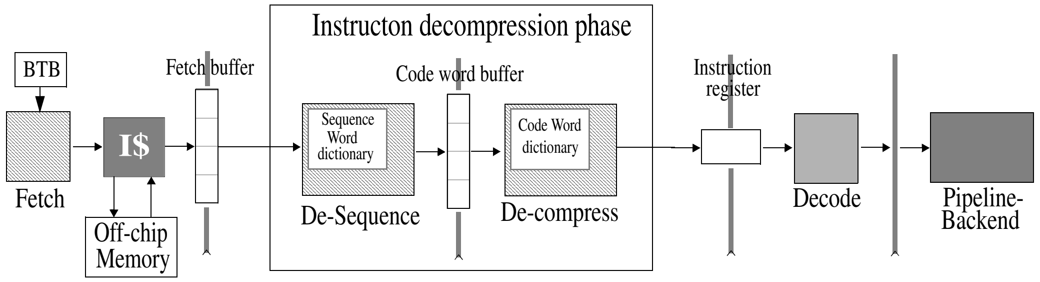


Fig. 1: Architectural overview of the expanded pipeline containing two extra stages.

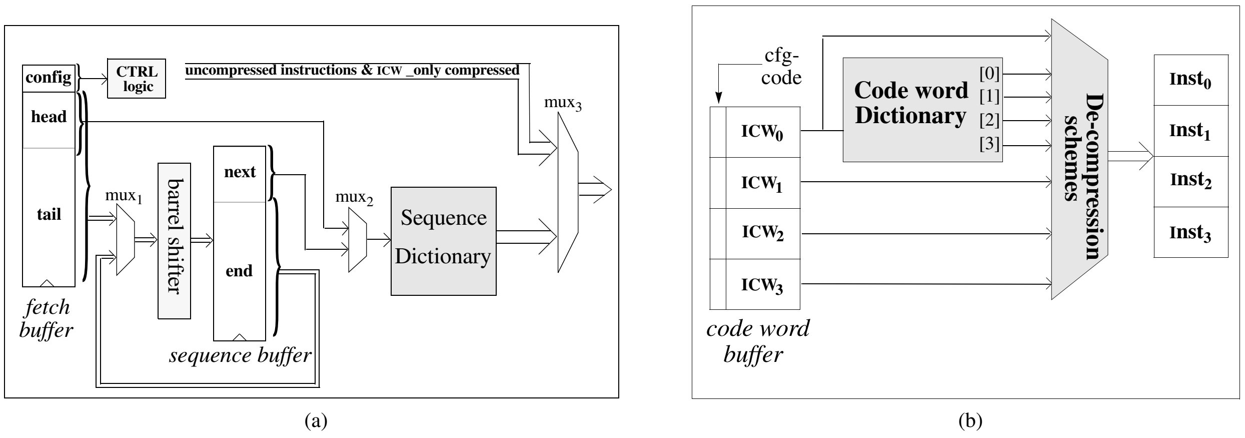


Fig. 2: Architectural sketches of the two decompression stages, a) Desequence, and b) Decompression.

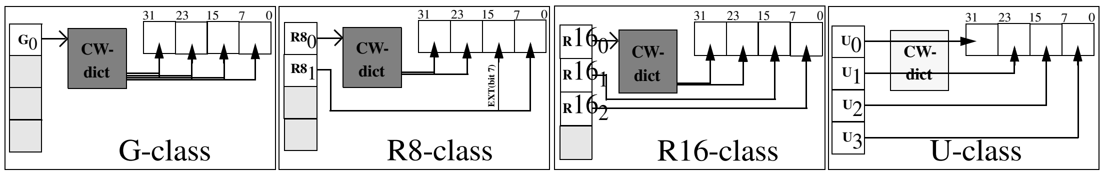


Fig. 3: The four decompression schemes used, one for each different instruction type; G-class, R8, R16, and the uncompressed U-class code word.

decompression is done in two stages, one could also envisage a one-stage approach leading to reduced branch-prediction miss penalty. Finally it is unclear how the compression scheme interacts with other methods with the same goal such as the filter cache. In this section we investigate some of these aspects.

1) Dictionary size: It is clear that with large dictionaries one can cover a larger percentage of the executed instructions in a program and thus compress more. However, a large dictionary also implies larger code words and therefore also less compression. However both the theoretical[1] and the practical[2] yield depend on more than just the number of code words used. Other important factors are: handling of branches, alignment policies, profiling methods, and compression algorithms or more precisely how data is handled and organized by the compression method.

[1]Theoretical or ideal, a measurements of the potential return with no regards to dynamic effects such as branch miss-predictions and speculatively executed instructions. This measurement can be estimated during the compression phase.

[2]Practical or real, measurements resulting from execution.

In section IV-B1 we present data that shows that we in practice can use a 64-entry sequence dictionary while the instruction dictionary has to contain at least 256 entries to be effective.

2) Cache organization: Caches can be organized in so many different ways. The first order design decision is of course cache size, but also address mapping mechanism (direct mapped or set-associative) and cache line size are important. From a compression point of view, dictionary code compression will make the instruction cache seem virtually larger since it will hold more instructions at any given time. The higher the compression ratio, the larger the virtual cache will seem. This will affect performance for smaller caches and to a lesser degree also energy which is more negatively affected by the higher fraction of capacity cache misses.

We present in section IV-B2 some results extracted from a large number of simulations with varying cache organizations and cache sizes.

3) Pipeline alternatives: Regarding the design of our processor pipeline, we consider a single-issue, in-order pipeline, but with speculative execution over predicted branches. We have chosen to put the Desequence and Decompression in separate stages, but they could just as well have been put in a joint stage instead. With a joint stage design we reduce the branch miss prediction penalty with one compared to the two-additional stage design. The two-stage design, however, enables a faster clock cycle time as the critical path in a given stage becomes shorter.

In order to evaluate the complexity of our design in more detail, we have made a VHDL implementation of the fetch, De-Sequencing and De-Compression functionality. The control logic was implemented as a finite state-machine and in order to make the timing analysis easier, all memories were implemented using D-flipflops. The design was then synthesized using the UCM 0.18 μm standard cell library. The synthesis results using Synopsys Design Compiler are shown in table I. Timing numbers were obtained using the UMC library with typical conditions. It is clear from the table that the area is not affected by having a the decompression done in one or two pipeline stages.

Although the test implementation of these pipeline stages shows that it is feasible to implement a joint Desequencing and Decompression stage, the impact on maximum clock cycle time is drastically affected possibly eradicating the gain from the shorter pipeline.The dynamic effects of this is investigated in section IV-B3.

4) Effect of Branch prediction accuracy: The branch predictor interacts subtly with the code compression mechanism. First of all, if the branch prediction accuracy is low and we get many wrongly speculative executions down the wrong path, the extra stages for the decompression will cause extra performance penalties. On the other hand, if we can align the compression so that a fetch word never crosses a basic block boundary, we only need to consult the branch predictor at the time when we actually need to go and get a new fetch word and thereby we save a considerable amount of energy in the branch predictor as well. Our compression method works this way.

We investigate five different branch predictor configurations and their effect on fetch, performance and energy ratios in section IV-B4.

TABLE I: VHDL design & ASIC implementation data of the Fetch, Desequence, and De-Compression stages.

10^3 **a.u.**	**Mem Area**	**Ctrl Area**	**Tot Area**	**Critical Path (ns)**
Single stage (joint)	1986	17	2003	3.28 (305 MHz)
Two stages (dis-joint)	1986	17	2003	2.43 (411 MHz)

5) Combinations with Filter Cache: Other techniques than code compression have been proposed to reduce the energy efficiency of the instruction fetch path, e.g. drowsy caches targeting leakage power consumption [**?**] and the filter cache [**?**]. These methods are orthogonal to what we propose here and it is worthwhile to investigate how they could be combined. We have in particular considered the use of a filter cache in conjunction with our two-level dictionary code compression scheme as shown in figure 4.

A filter cache acts as an L0 cache and filters out the currently most accessed instructions and stores them in a small, fast and energy efficient cache like structure close to the processor core. The size of a filter cache range between a few instructions to about 256, about equal to 8-1024 bytes. The result is fewer ordinary instruction L1 cache accesses that cost more power to access With this scheme there is an extra access time penalty associated with the extra L1 access. However, there are schemes where both filter cache and L1 are accessed in parallel which do not have this time penalty but then you loose the energy efficiency. With code compression and its property to increase the density of the code, higher density virtually increases the capacity of the cache. This property also applies to the filter cache which either could, with constant size contain more instructions or with reduced size contain the same number of instructions, compared to an architecture without compression.

One could also consider a scheme with a BTB-directed filter cache which reduces the overhead of being forced to store duplicated tag entries for identification of the FW that's been stored in the buffer, see figure 4 (c).

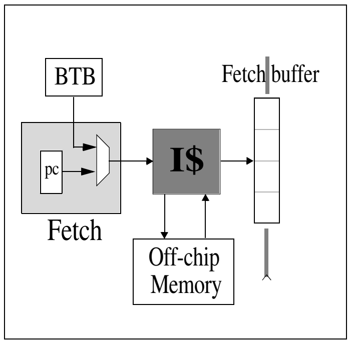
At the time of this writing, we have not yet finished analyzing the results from measurement on combinations of filter caches and the two-level code compression scheme. However, preliminary results show that the effects are cumulative and thus work in synergy.

IV. Experiments

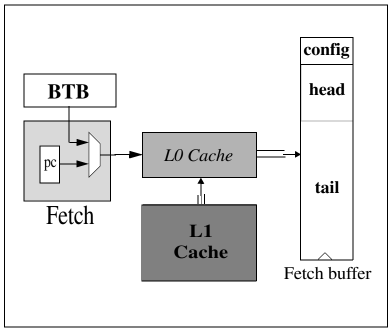
A. Methodology and workload

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 [**?**], [**?**]. Additional pipeline stage(s), buffers, and dictionaries were added to the simulation model as needed. To model energy for off-chip memory accesses, the simulator was augmented with an energy model for memory and bus interface based on the IRAM project [**?**]. We also made an implementation of the one-level approach for comparison purposes [**?**] as well as a model of a standard five-stage pipelined processor. Simulation parameters for the three different processors can be viewed in table II.

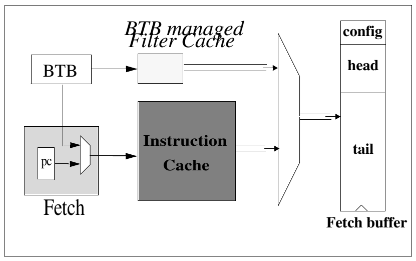
We have used the MediaBench benchmark suite to evaluate the effectiveness of our proposed compression scheme [**?**]. They were compiled without modification with gcc version 3.2.2 with optimization level -O2.



(a)



(b)



(c)

Fig. 4: Three different filter cache combinations: (a) no filter cache is used, all fetch words are fetched directly from the instruction cache or main memory. In (b) the traditional filter cache is shown while in (c) we have introduced a btb-controlled filter cache.

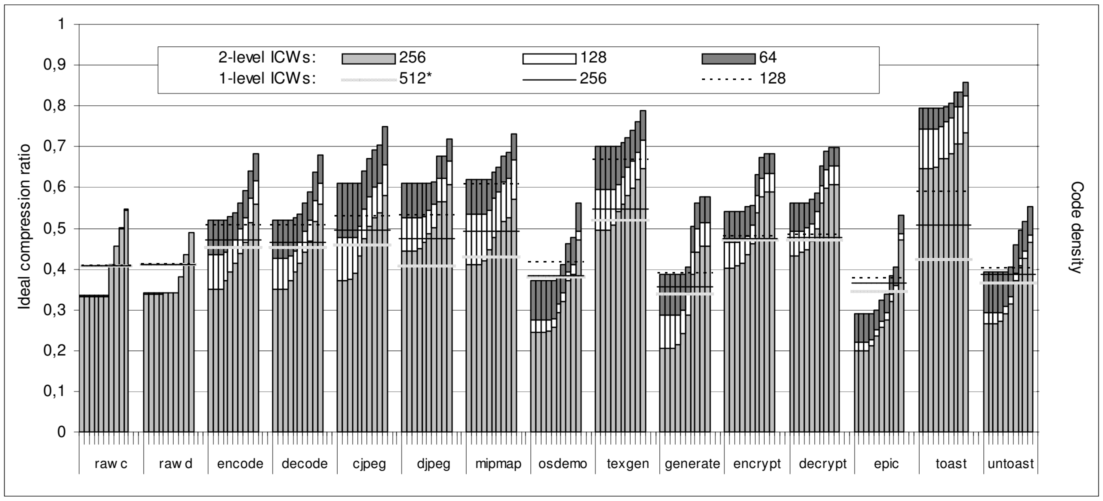


Fig. 5: Ideal dynamic compression ratio and code density for the different dictionary configurations available to the two code compression architecture. The different columns for each benchmark represent a different sequence dictionary size ranging from 256, 128, 64, 32, 16, 8, 4, 2, 1, to 0.

TABLE II: Processor simulation parameters.

Processor:	
Mispredict penalty	Baseline: 3, 1-level: 4, 2-level: 5
2-bit bimodal branch predictor	1024 entries
Branch target buffer (BTB)	128 entries, direct mapped
Return stack	8 entries
Memory system:	
L1 I-cache	16 KB, 4-way, 32 byte line size, 1 cycle latency
L1 D-cache	16 KB, 4-way, 32 byte line size, 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
$V_d d$	1.8 V
Clock frequency	400 MHz

B. Results

1) Dictionary size: The graph in figure 5 shows the ideal dynamic compression ratio and the ideal code density, which is the inverse to compression ratio and gives a mean value of how many instructions it is possible to get on each fetch. These values have been calculated as part of the code word selection process and are based on profile data only and not on architectural simulations. Therefore they represent the ideal dynamic compression ratio.

The different columns for each application represent a sequence dictionary configuration with 256 entries down to 0 in steps of 2-1. Each segment of the individual columns represents the numbers of entries used in the instruction code word dictionary. Also in the figure, represented as lines, are the ideal results corresponding to using a one-level approach (i.e. not being able to encode code sequences) using different number of code words, 512, 256, 128. The two-level approach makes use of two dictionaries which increases the total storage, both dictionaries can contain up to 256 entries. Therefore, in order to make a fair comparison of the potential yield, we introduce a hypothetical one-level configuration with 512 code word entries denoted 512*. This would affect the code word size negatively which in turn would impact on compression, but that is for our comparison purpose here ignored.

From figure 5 it is clear that increasing the number of code words used for the one-level architecture yields a noticeable improvement on code density, except for the rawc—d applications which have quite small working sets. The return is however diminishing for each time the dictionary size is doubled. Looking at the 2-level configurations, reducing the number of sequence dictionary entries will at some point impact on code density. For a majority of the applications this point appears at around 128-64 entries. Focusing on instruction code words we can see that the reducing the number of code words have a much more dramatic effect on code density. Due to this observation and our preliminary experiments [**?**], we can say that despite using two dictionaries, the most important factor for high compression ratio still is the size of the code word dictionary.

Worth to notice and a part of our motivation for the presented 2-level approach, is the fact that adding a small sequence dictionary will indeed improve on density for a majority of the applications. This is especially noticeable for those applications with none or a rather small improvement in code density when the size of the level-1 code word dictionary is doubled, consider rawc—d, encode, decode, osdemo, epic, and untoast. Other applications, for example djpeg, texgen, and decrypt are more dependent on a large quantity of instruction code words. A special case is represented by toast where a single commonly executed basic block of 401 instructions prevents effective compression because it would require more instruction code words than what we have available for this basic block only.

For most of the applications using 256 entries in the instruction dictionary is essential as any number less than 256 would severely affect compression ratio. Regarding the sequence dictionary most applications can make do with only 64 entries without serious degradation in compression ratio.

2) Cache organization: Figure 6 (a) shows the performance of all applications for two-way set associative address mapping for different instruction cache sizes ranging from 16 KB down to 1 KB. Performance is normalized to no compression and 16 KB cache size. For each application/cache size combination there are three bars: level_2 representing two level code compression, which is the main alternative in this paper, level_1 representing one-level dictionary code compression, and level_0 which is no compression at all.

It is clear that for the larger cache sizes, there is no performance advantage of having code compression. In fact, there is a small disadvantage as the longer pipeline (two extra stages) lead to higher branch prediction miss penalty. However, when the caches are small, there is a substantial advantage of having code compression as this virtually increases the cache size.

Figure 6 (b) shows the same thing but for energy. The comparative analysis is here a little bit more complex. First of all, the energy bars have a bath tub shape with varying cache sizes. When reducing the cache size, the energy first goes down then increases again. The reason for this is that smaller caches draw less power and if it does not results in excessive number of cache misses, this is only good. However, when the cache becomes small enough, the energy from going off chip out-weighs the advantages.

We have measurement data from many more cache line sizes and several other address mapping schemes, but the trend for these measurements are the same as in the ones presented here.

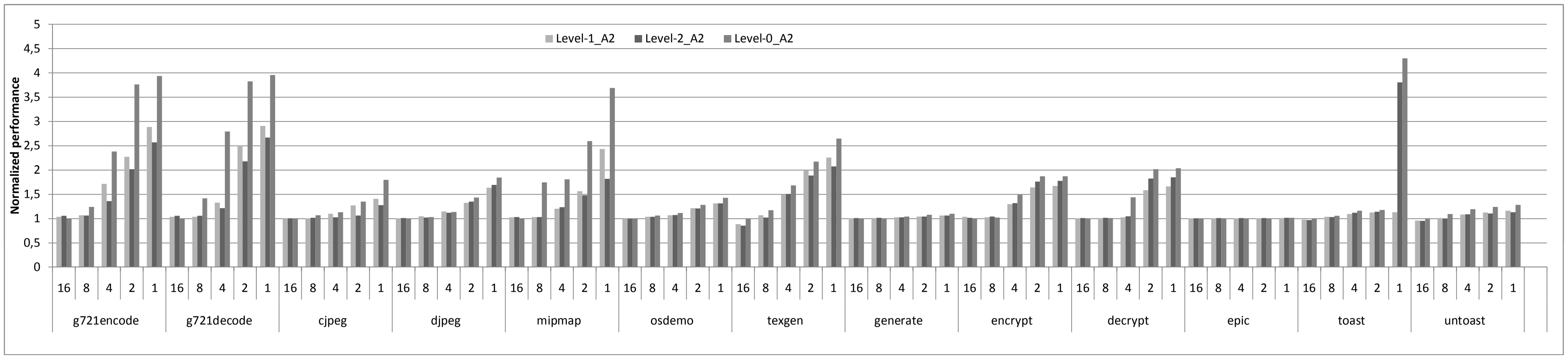
3) Pipeline alternatives: To analyze the dynamic effects due to pipeline design, cycle time and miss prediction penalty, a small experiment was made with a few of our test applications that corresponds to a representative behavior of the Media-bench applications. The result presented in figure 7 shows normalized performance and total energy as well as fetch path energy-ratio broken down into its components for five different architectures. Besides our base-line and one-level architecture we here also consider three 2-level configurations, disjoint_400, joint_300, and joint_400, where the name refer to pipeline depth and clock speed in MHz. Although not feasible, the joint_400 represent a hypothetical implementation included for comparison.

From our previous studies [**?**], [**?**] we have seen that there almost exist a one-to-one relation between the reduced access ratio of I-cache and the energy reduction in I-cache. However comparing the energy ratio for the I-cache in figure 7 with the ideal compression ratio presented in figure 5, this relation does not hold. This is due to dynamic effects, particularly speculative execution down the wrong path as a consequence of branch miss-prediction. The deeper the pipeline, and the worse the branch prediction accuracy is, it is not only a performance factor but also a factor directly coupled to compression and energy. The impact on performance due to the longer disjoint decompression is very small under the presumption that branch prediction accuracy is good. Here both rawdaudio and encode experience a rather low accuracy of about 70-85%. For those applications the base-line processor is about 3-6% faster that the single stage architectures and about 6-12% faster that the two stage design.

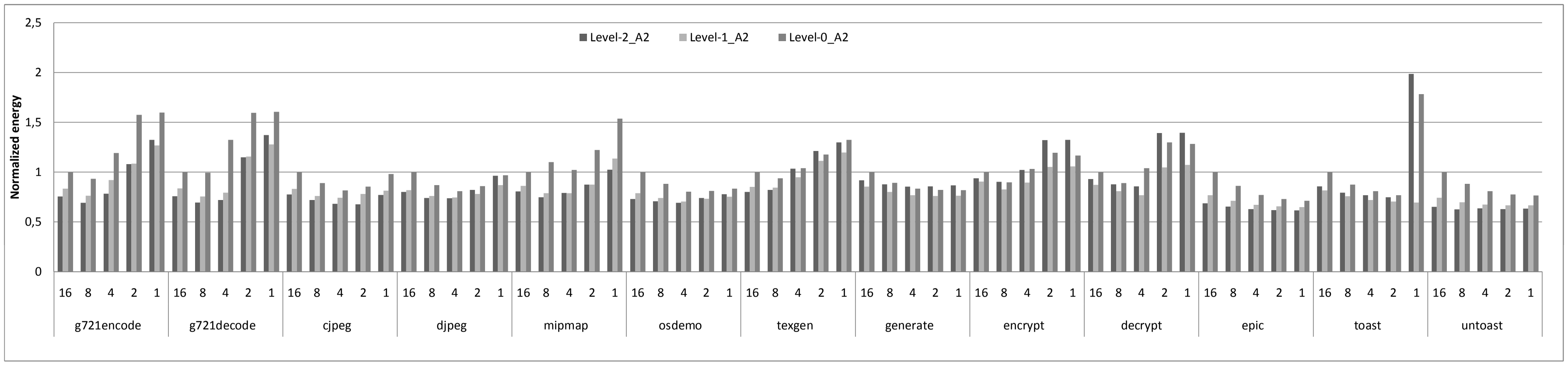
The 300 MHz joint pipeline design consumes on average 32% less energy in order to execute the instructions, however due to the longer execution time of about 33% compared to the disjoint_400, this gain is not translated into energy savings but rather an average increase of 4% in total energy consumption.

4) Effect of Branch prediction accuracy: Figure 8 (a) shows the effect of various branch predictors with different accuracy properties. The selected applications represent a cross-section over the properties of all Mediabench programs. The selection is based on the number of executed instructions, encode being the largest and rawdaudio the smallest. Figure 8 (b) shows the branch/jump statistics for the same applications.

The experiment comprises of a series of simulations of the disjoint_400 pipeline with a 256_256 dictionary configuration.



(a)



(b)

Fig. 6: Performance (a) and energy (b) normalized to 16 KB cache, no compression, for various cache sizes and applications.

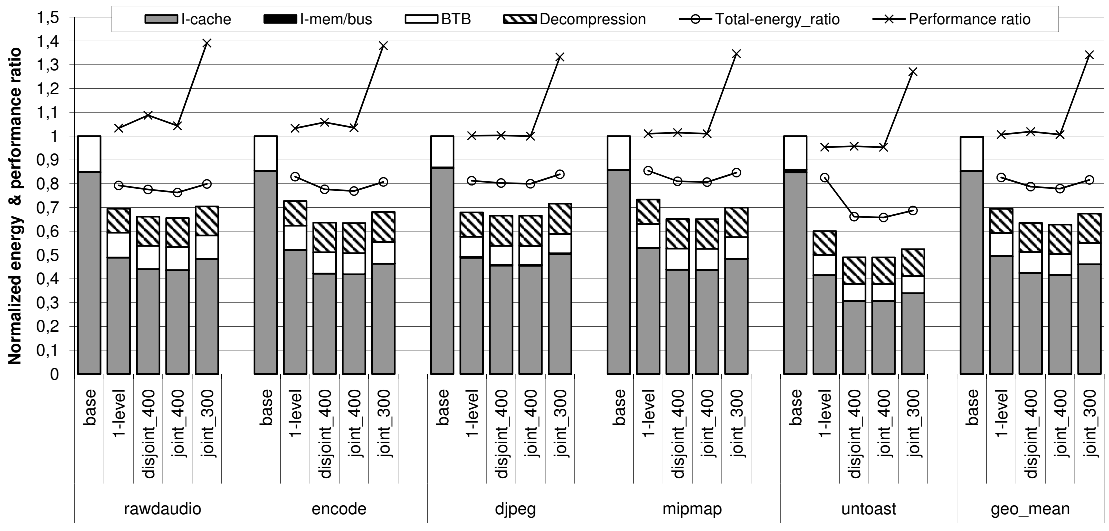


Fig. 7: An analysis of performance and energy for our base-line and compression architectures. Notice that for the 2-level architectures we use a 256_256 dictionary configuration.

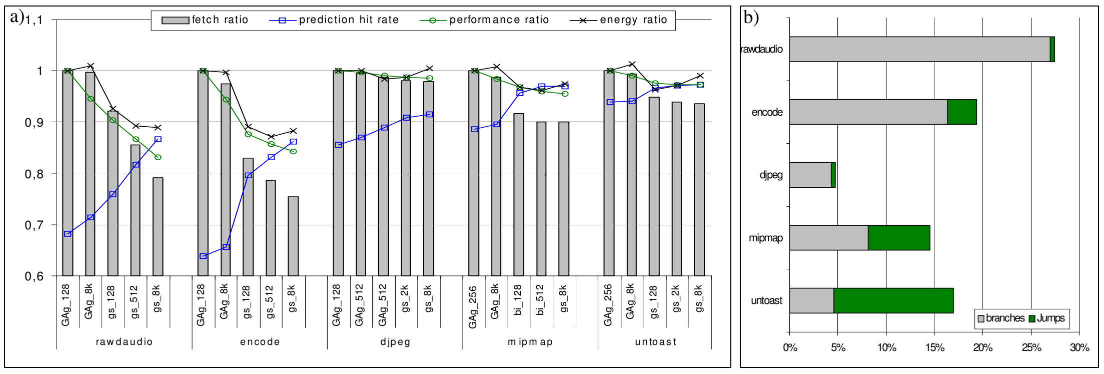


Fig. 8: (a) Impact of branch prediction accuracy on performance, energy, and fetch ratio. (b) The share and distribution of branch & jumps instructions executed by the different application.

As we only are interested in monitoring the effects on compression ratio and performance as a function of prediction accuracy, we do not consider the feasibility of the occasional use of staggeringly large and complex predictors. The results shown in figure 8 (a) are arranged with regards to prediction accuracy. Therefore each predictor, left to right, represent a predictor that delivers consistently better accuracy.

This small experiment clearly indicates that prediction accuracy indeed does affect the compression ratio. It is most visible for rawdaudio and encode as those applications display the largest sensitivity to prediction accuracy. We can also confirm the established truth that performance is highly dependent on prediction accuracy. .

V. Conclusions

In this paper, we have investigated design alternatives and their implications for a two-level dictionary code compression method. Based on a what 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.

Several of the aspects presented in this paper have never been presented before. No-one has in such detail evaluated architectural implications for dictionary code compression schemes. In particular we have implemented critical parts of our design in VHDL to test the feasibility of our design.

The other main conclusions to bring home from this study are:

- *Dictionary code compression works best for small caches.* Small caches has the benefit that access energy is low and the performance effects of compression is positive.

- *Branch predictors must be good.* The dictionary code compression works poorly if the branch predictor is poor. On the other hand, our scheme improves on branch predictor energy consumption so the cost of having a good predictor might not be that high.

We have not yet investigated the compiler optimization interaction with code compression and other kinds of compiler support. There are many things to improve in these areas. In particular we believe that it should be possible to break the large basic block of toast in order to make it benefit from the 2-level code compression scheme as well.

Acknowledgment

Mats Brorsson is a member of the HiPEAC European network of Excellence (http://www.hipeac.net).