Adaptive and Flexible Dictionary Code Compression for Embedded Applications

Mats Brorsson and Mikael Collin

KTH ICT Dept. of Electronic, Computer and Software Systems
Electrum 229, SE-164 40 Kista, Sweden
email: {Mats.Brorsson, Mikael.Collin}@imit.kth.se

ABSTRACT

Dictionary code compression is a technique where long instructions in the memory are replaced with shorter code words used as index in a table to look up the original instructions. We present a new view of dictionary code compression for moderately high-performance processors for embedded applications. Previous work with dictionary code compression has shown decent performance and energy savings results which we verify with our own measurement that are more thorough than previously published.

We also augment previous work with a more thorough analysis on the effects of cache and line size changes. In addition, we introduce the concept of aggregated profiling to allow for two or more programs to share the same dictionary contents. Finally, we also introduce dynamic dictionaries where the dictionary contents is considered to be part of the context of a process and show that the performance overhead of reloading the dictionary contents on a context switch is negligible while on the same time we can save considerable energy with a more specialized dictionary contents.

Categories and Subject Descriptors

C.1.1. [**Single Data Stream Architectures**]

General terms

Measurement, Performance, Design

Keywords

Dictionary code compression, Instruction profiling, Processor architecture, Instruction memory bandwidth, Fetch path energy

1. INTRODUCTION

As the market for consumer electronics relentlessly continue to grow, the vast amount of processors in the world are found in embedded applications. The gap between what typically has been defined as desktop applications and embedded applications is also diminishing. New novel embedded and hand-held products, mobile phones, digital cameras, pocketPCs, PDAs and portable game consoles, offer multimedia capabilities and high performance communication facilities. These all require sophisticated hardware resources and elaborate general purpose software. Yet, embedded applications pose challenges such as limited resources of different kinds. Examples of these requirements are limited amount of memory, limited power budget, the need for energy conservation for battery-operated devices and a small form factor.

However, with the escalated complexity of embedded software, application code size tends to grow. Larger and more advanced programs require both larger sized memories and higher performance and as a consequence this leads to higher power and energy consumption. It is therefore imperative that low power and energy consumption is considered early and in all design stages.

In this paper we are focusing on the instruction fetch path of moderately high-performance 32-bit RISC processors for embedded applications. We are presenting a working complete solution for an instruction code compression scheme that achieves compact code, reduced energy consumption, and smaller chip area with preserved performance as compared to a baseline system for a range of typical embedded programs. The scheme is based on full-length 32-bit RISC instructions being replaced by the compiler, based on a previously generated execution profile, to shorter code-words. These code-words are during execution looked up in a dictionary in an extra pipeline stage where they are substituted by the original 32-bit instruction for rest of the pipeline execution. Figure 1 shows an outline of the architectural solution. The fetch stage (F) accesses the instruction cache and retrieves a *fetch unit* of one word. With uncompressed instructions this is exactly one instruction, with compressed instructions it can contain up to four code words representing four instructions. The fetch stage puts the fetch units in a buffer which can contain up to two fetch units. The decompress stage (DP) reads from this buffer, determines whether the first element in the buffer is a compressed instruction or not and if so, decompresses it using the dictionary and finally puts it into a queue for the rest of the pipeline which is identical to a processor without code compression.

The reason for looking in particular at the instruction fetch path is that when it comes to energy consumption, the instruction cache has a significant and elevated role in the light of the processor-memory interface being a significant and large contributor to the whole system energy consumption [18, 20]. The impact of instruction fetches is further underlined considering that the instruction cache is the most accessed memory structure. Albera

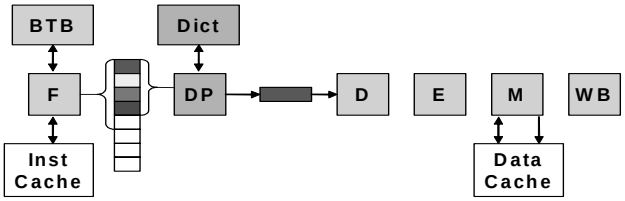


Figure 1: Outline of the proposed processor pipeline, including decompression stage and dictionary table.

CASES'06, October 23–25, 2006, Seoul, Korea.

and Bahar [1], shows that more energy is consumed by the instruction cache than in the data cache due to the higher access rate to the instruction cache.

These observations were also made in a feasibility study of a dictionary code compression scheme reported in [7]. This study indicates that about 25% of the total system energy stems from the instruction fetch path. Breakdown of the energy spent during instruction fetch reveals that 75% originates from the instruction cache and the majority of the rest from the branch prediction logic since its tables are also accessed on every instruction fetch.

Dictionary-based code compression has been proposed before (see section 2 for related work descriptions), but in contrast to previously published results we have taken a holistic approach showing solutions for all problems in the entire chain of the system. In particular, our contributions are:

- *A flexible and adaptive code-word dictionary lookup scheme*. Both static and dynamic dictionaries are investigated. In the latter case, the contents of the code-word dictionary is considered to be part of a process' context. This is the first time dynamic dictionaries are proposed and evaluated.
- The development of an *innovative profile process*. Previous work has not taken aggregated profiles over groups of programs into account. In contrast, we have a process to aggregate a profile based on a previous profile-execution of one or more executables.
- Development of a working *instruction set extension* preserving the existing 32-bit instruction set in coexistence with short code-words and a *compiler algorithm for generating executables* using the execution profile. Previous work has not proposed integrating this in the compiler making the whole process a little bit awkward.
- A *complete microarchitecture proposal* implementing the architectural support needed for the proposed scheme integrated in the front-end of the processor. Most other proposals neglect this part making it difficult to assess the final effectiveness or effects on performance and power consumption. Part of the architectural support needed in an earlier feasibility study has been synthesized and checked for performance bottlenecks [7, 17].
- A *thorough evaluation* using MediaBench [11] benchmarks where area, performance and power overheads of the proposed scheme is taken into account. Again, this is in many cases neglected in prior art.
- An *architectural exploration* showing the consequences and possibilities with smaller instruction caches and cache line size. Previous work has shown the possibilities to reduce the cache size, but none has shown on the possibility to also reduce the cache line size when code compression is used.

Experiments using the Mediabench applications [11] show that we achieve a static compression ratio is between 0.67 and 0.79 with an average of 0.75. This is in line with previous results. The instruction cache access ratio is between 0.40 and 0.55 resulting in a reduction in the fetch path energy consumption of 30% to 50%. This translates to roughly 20% reduction in overall energy consumption. With smaller caches the overall energy savings are higher since the miss rate goes up. Performance-wise we show that for some applications, we can reduce the instruction cache size with 50% and still have the same performance-level while still saving 20% energy. Compared to the baseline architecture, for which we cannot reduce the instruction cache for performance reasons, we can achieve up to 30% of total energy savings with a smaller instruction cache with a 3% performance degradation.

The rest of the paper is organized as follows: After an account of the most important related work we continue with a description of the profiling process. The code generation process, based on the profile, is described in section 4 followed by a description of the microarchitecture. We then show the effectiveness of our approach in section 6 followed by conclusions.

2. RELATED WORK

Several researchers and/or companies have worked with instruction memory compressions or short instructions, e.g. [10, 14, 16, 19, 21]. Other researchers have, as we, worked with *dictionary compression* where instructions in the code are replaced by a short code word. The code word acts as an index to a dictionary holding the information needed to restore the original replaced instruction. Decompression hereby just becomes a lookup in a small and fast table.

In [12], Lefurgy et al., propose a dictionary compression technique with the objective to reduce static code size in memory. Based on static occurrence frequency, common sequences of instructions are encoded and substituted for short code words in the object code. Depending on the native ISA, using variable length code words of 8, 12, or 16 bits, an average static compression ratio between 0.61 to 0.74 is reported for a set of CINT95 applications. No dynamic, performance or energy measurements are reported. Independent of the choice of ISA or whether fixed or variable length code words are used, they conclude that the most important factor for high compression ratio is the dictionary size.

In [13] Lekatsas et al., present a dictionary compression method and the design of a fast, one-cycle decompression unit. In their solution they make use of variable-length code words, of 8 and 16-bits, to compress 24-bit native instructions. An average dynamic compression ratio of 0.65 is reported resulting in a 25% increase in performance on average. However, they used artificially small cache memories forcing the miss rate high and the doubled table will increase the energy consumption for which they present no results.

Another interesting dictionary compression method is presented in [15]. Individual instructions are compressed into 32 bit long *ComPackets*. In the paper the authors discuss and present an interesting approach for selecting dictionary contents where they mix static and dynamic profiling. On a set of MiBench and MediaBench applications [9, 11], an average compression ratio of 0.75 was achieved. Simulations shows an average reduction in execution time by 27% and an average 46% reduction of instruction cache energy. Just as Lekatsas et al., they use unrealistically small instruction caches and they do not present any microarchitecture solution for their decompression engine so performance results for the entire system cannot be presented.

In [2], Benini et al. presents three compression schemes that significantly trade off code size for performance and memory traffic reduction. Their best solution is based on an having a 255-entry dictionary, containing the most frequently executed distinct instructions. The code is organized in a manner having compressed- and uncompressed code located in separate memory sections. Each instruction in the program is compressed into 8-bit

code words and stored in the compressed memory section. For all instructions not present in the dictionary the non-existing 256th index is used, indicating an uncompressed instruction which must be fetched from the uncompressed memory section, yielding an additional memory reference. The uncompressed section contains all original program instructions. This technique obviously do not improve on the static memory required. On average, including the dictionary, 27% more memory space is needed, storing both the entire original and the compressed program. This increase is motivated by positive dynamic advantages. Experiments performed with a model of a small pipelined processor, show that using compression compared to only execution native instructions dynamic memory ratio of average 0.35 is possible to achieve, resulting in an significant 0.40 in fetch energy ratio.

A major complication in the design of a code compression scheme is how branches are handled. If a compressed branch instruction ends in the middle of a fetch unit, unused bytes are fetched from the instruction cache. If a branch target is not aligned to the start of a fetch unit, unused bytes are again fetched. Also, the actual compression of branch instructions is non-trivial because of the new address space used in the program. Most solutions mentioned above do not describe this process satisfactorily or not at all.

3. PROFILING

In this section we describe the profiling process used, how we can aggregate the profiles of several executions and some results for the MediaBench programs [11]. For the purpose of this study we profile the execution of distinct instructions. Two instructions are considered distinct if their corresponding bit-pattern differ in at least one bit position. Thus two add-instructions are different if they have different arguments.

3.1 The profiler

We are using a simulation-based profiler to gather instruction execution frequencies. Although we have used simulation, it is possible to use statistical sampling from executions on real hardware. In [3] the authors present a very low-overhead sampling technique which they apply for instruction and data cache miss predictions. The same technique can be used also for instruction profiling.

Figure 2 shows an example of the output generated by the profiling tool for the program pegwit encrypt from MediaBench [11]. The output file contains all distinct instructions, bit-patterns, that were encountered during the entire program execution. The entries are sorted in descending order based on their frequency of use. Each entry consists of a 64-bit hexadecimal *bit-pattern*, the number of times that instruction was *executed*, and the *pattern coverage*. The pattern-coverage is the fraction of which that actual instruction contributes with in respect to the total amount of instructions executed when the profile was generated.

encrypt.pat			
i	bit-pattern	executed instructions	pattern-coverage
1	4200000003020300	767158	0.022621
2	5200000002030200	751929	0.022172
3	a200000000003ffff	739701	0.021811
4	5100000003030002	739701	0.021811
5	5500000000040201	730012	0.021525
6	2600000003030000	728941	0.021494
7	5500000000020201	503881	0.014858
8	4f0000000206ffff	420196	0.012390

Figure 2: Profiling output file format

We are using two different types of coverage in this paper: *profile-coverage* and *execution-coverage*. The profile-coverage is a measure of how large fraction of the profiled program's distinct instructions is covered by the resulting profile. It is calculated as the sum of the pattern-coverages for the number of instructions that are going to be part of the profile. Execution coverage is related but specifies how large fraction of an execution can be covered with a particular profile, which may not be based on the executed program.

3.2 Profiling detail

A novel aspect of this paper is our ability to aggregate profiles for two or more executables. We are using the MediaBench [11] suite which consists of a number of applications that typically have two or three components, or functionalities, for example, encoding and decoding. To make some sort of coverage over the broad range of workload configuration possible, we generate three different levels of detail of profiles:

1 Functionality *funct*: The most specific of all profile methods available. The profile is based on all instructions executed when a special functionality of an application is executed.
2 Application *appli*: This profile is based on the execution of the functionalities within an application. For the case of jpeg, the profile is an aggregation of the most executed instructions by both cjpeg and djpeg.
3 MediaBench *bench*: The least specific of all profiles. The profile is an aggregation of the most frequently executed instruction patterns over all the different MediaBench applications used in this study.

The profiler generates funct-profiles. To generate an appli-profile, the profiler takes all funct-profiles belonging to an application as input and generates a combined profile. For the jpeg application, the cjpeg- and djpeg profiles would be constitute the input files.

The bench profile is in this context an aggregation of all programs from the MediaBench suite used in this study. A generic number of appli-profiles are fed as input to the profiler which generates a profile based on all executed instructions over all applications employed.

3.3 Profiling results

We have through simulation with the profiler obtained the three level of profiles as described above. In the first, the funct-profile, every functionality of the applications gets a unique profile tailormade for that special functionality. In the second, we get a a profile for each application, appli. And finally in the bench-profile, we get a profile covering all executables in MediaBench.

One important aspect of the dictionary code compression scheme is its ability to reduce the instruction memory bandwidth requirement. When the possible coverage using a specific profile of size n, is known, the possible improvement on required instruction fetch bandwidth can be estimated using the model described in equation 1.

$$\text{improved bandwidth} = 1 - \left((1 - \text{coverage}) + \frac{\text{coverage}}{4} \right) \quad \text{(eq. 1)}$$

If we want to achieve a 50% reduction in bandwidth requirement, this model tells us that we need a coverage of 67%. This profiling study therefore aims at distinguish how large set of patterns is needed to obtain a coverage of 67% or higher. The selected pat-

Table 1: The number of distinct instructions in the three levels of profiles separated whether library instructions are included (w. lib) or not (no.lib).

Benchmark		#Executed instr (10^6)	Library instr.	Funct-profile		Appli-profile		Bench-profile	
				#Dist. instr. / Profile		#Dist. instr. / Profile		#Dist. instr. / Profile	
Appl.	Function			w. lib	no. lib	w. lib	no. lib	w. lib	no. lib
adpcm	rawcaudio	6.6	0.2%	1098	134	1268	211	27722	22911
	rawdaudio	5.5	0.3%	1078	112				
g721	encode	335.9	8.8%	1591	624	1902	781		
	decode	337.9	11.8%	1616	616				
jpeg	cjpeg	15.8	1.7%	4065	3176	6646	5384		
	djpeg	4.8	5.6%	4320	3190				
mesa	mipmap	83.4	61.7%	7036	5645	12848	10813		
	osdemo	28.4	68.0%	6846	5424				
	texgen	128.0	43.7%	8499	6951				
pegwit	generate	13.2	2.0%	2784	1892	4137	3059		
	encrypt	34.5	2.3%	3817	2804				
	decrypt	19.6	2.3%	3788	2782				
epic	epic	55.6	4.5%	3823	1323	4673	1941		
	unepic	7.5	9.4%	3165	901				
gsm	toast	225.5	0.3%	3572	2420	4181	3031		
	untoast	63.1	1.2%	2602	1359				

terns are later used to produce the dictionary in the decompression stage in the pipeline. Therefore the number of patterns must be kept at a feasible size in order not to result in a too large and thereby slow dictionary that impedes on cycle time. Also, as the number of used patterns increases the number of bits needed to index the dictionary it will reduce the number bits available for the remaining instructions. A feasible number of patterns ought to be a power of 2, between 16 and 256.

Table 1 shows the number of distinct instructions in the three levels of profiles for all MediaBench applications and functions. We have also conducted an experiment to study the impact of library code on the creation of the profile. For most programs, the fraction of library code is from negligible to less than 10%. However, for the mesa application, the fraction of executed instructions that are from the library is significant. The difference of profiling with the library or not, in terms of the number of distinct instructions, is smaller with larger scope of the profile. On the other hand, the total number of distinct instructions increases greatly which results in a smaller fraction of the profile that can fit in the dictionary.

Figure 3 shows the execution coverage for programs when the funct-profile of each functionality is used. The columns are divided into segments showing the coverage for 16, 32, 64, 128 and 256 distinct instructions in the profile, respectively. In summary, all programs reach the 67% coverage with 256 distinct instructions with an average coverage of 89%. 11 out of the 16 functionalities reach the goal with only 128 distinct instructions.

Table 1 also contains the number of distinct instructions for aggregation of functionalities into application profiles. The contribution of each functionality is proportional to each one of the contributing functions share of the total amount of instructions executed during generation of the profile.

Each programs execution coverage, for the different appli-profiles produced is presented in Figure 4. Although the coverage is lower than for the more specialized profiles, we still get relatively good results indicating that, at least for some programs, the func-

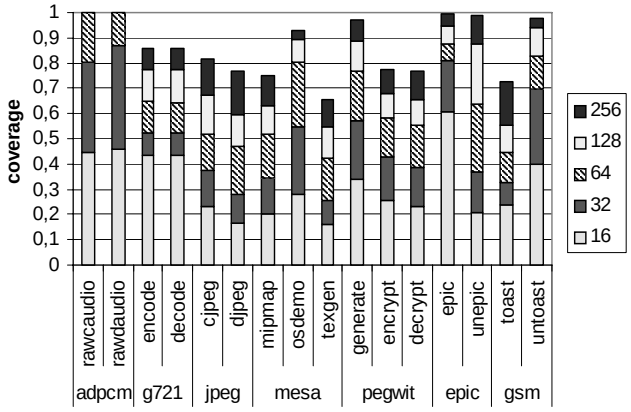


Figure 3: Execution-coverage for the different functionalities using their funct-profiles.

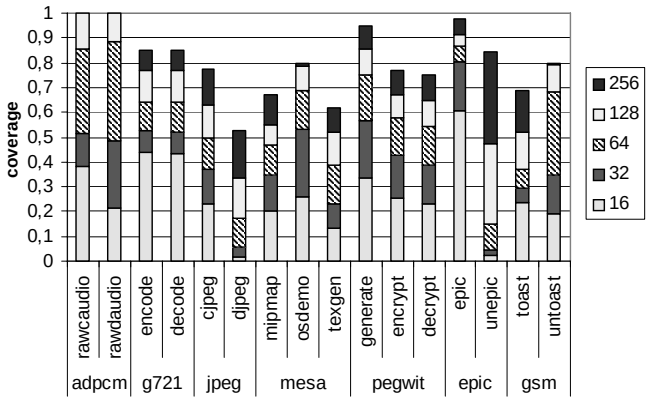


Figure 4: Execution coverage for the individual functionalities experienced when executed using their appli-profiles.

tionalities contributing to an application share large portions of the set of executed distinct instructions.

To summarize the results, the most obvious observation is the overall reduction in peek coverage for all the individual functionalities. Also, the number of patterns needed in order to reach the desired 0.67 in coverage are in some occasions increased compared to when all applications use their funct-profiles. Despite the overall reduced yield, and sometimes very severe reduction in coverage for some individual applications the results considering the appli-profiles are positive. That is important, in respect to that the aggregated appli-profiles constitutes the "lowest common denominator" for the case when all functionalities within an application are to be executed.

As described before the bench-profile is the least specific of all the profiles made. Figure 5 shows the execution coverage for all programs, and the aggregated execution coverage for the entire MediaBench set when the bench-profile is used. Of all the profiled instructions, only 58.8% can be covered using the profile, when 256 patterns are used. The corresponding number for using 128 patterns is 46.3%, see the left column of Figure 5.

Summarizing the results regarding the usage of bench-profile, one significant conclusion can be made. The bench-profile is too general and broad, trying to incorporate too many different functionalities. The contributing applications are too divergent in their usage of instructions. Looking at what levels of coverage are possible to obtain for the individual functionalities the result is weak. Only the largest contributor to the profile shows an acceptable behavior, due to the objective to promote high coverage for the functionality that contributes the most to the profile. All other functionalities depends on being as similar to the dominating functionality as possible in order to achieve high coverage.

Using the bench-profile as a general profile is not applicable. The bench-profile should only be considered to be used if the workload really comprises of all the profiled functionalities and all are executed equal amount of times. The profile used should, for best yield, be based on the specific workload in mind. For multi application workloads especially with an unbalanced relation between the execution frequency of the applications, execution frequency should be taken in to account, giving the more frequently executed functionalities higher significance in the profile.

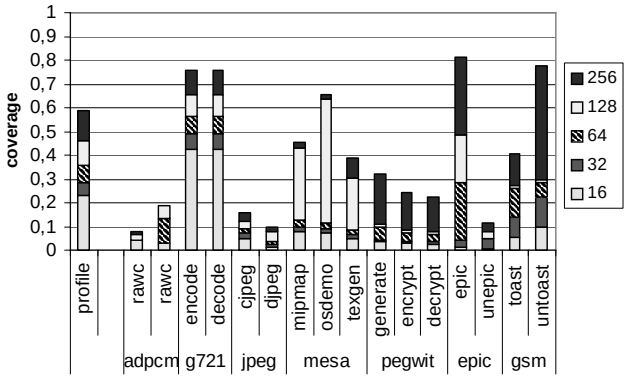


Figure 5: Execution coverage using the bench-profile.

4. COMPRESSED CODE GENERATION

The concept behind the presented code compression scheme is to encode the most frequently executed instruction in a shorter format in order to reduce the traffic to the instruction cache and thereby achieve better performance for smaller caches and greatly improved energy efficiency. The key idea of the scheme is shown in Figure 6. The instruction sequences {B,C,D} and {G,H,I,J,K} constitutes loop constructions, possibly iterated several times. During execution, the instructions inside the loops are possibly more frequently executed than instructions located in non-iterated sequential sections. When we apply compression on the most frequently executed instructions, the number of bytes fetched in order to execute the code sequence is reduced as more information is acquired on each fetch, assuming a fetch unit of 32 bits. The density of the code has improved, which will affect the processor-memory interface, improving presumably on both performance and energy consumption. As shown in Figure 6, after profiling, the bit-patterns of the eight instructions in the two loops are each assigned and placed in a dictionary entry. Branch instructions are marked with an extra encoding bit in the dictionary, indicated by the black marker in the figure. In the program code each occurrence of a dictionary instruction is then substituted for a short dictionary *code word*, corresponding to the index position of the dictionary entry containing the bit-pattern of the substituted instruction.

4.1 Branch instructions

Compression, translation and execution of non-path-altering instructions, add, load and alike is rather straight-forward according to the selected compression policy. The situation regarding path-altering instructions is somewhat more elaborate, especially when instructions of different lengths are allowed to be mixed interchangeably. The issues regarding branches and jumps are widely debated in the research literature [2, 12, 13, 14, 15, 21].

The dilemma of how to deal with and handle the implications imposed by branch instructions can roughly be categorized into three not entirely disjoint issues, as they influence on each other.

Address translation: Absolute and relative addresses found in the code are no longer valid after compression as they no longer point out the correct instructions. Solutions to this problem involves usage of translation tables, or re-calculation of target addresses with patching of the code afterwards, as the exact position of the target is not known prior the compression. Our approach is to include this in the compilation phase where we know the exact lengths of the instructions.

Alignment: It is important to align branch targets to the start of a fetch unit as the instruction(s) before the branch target other-

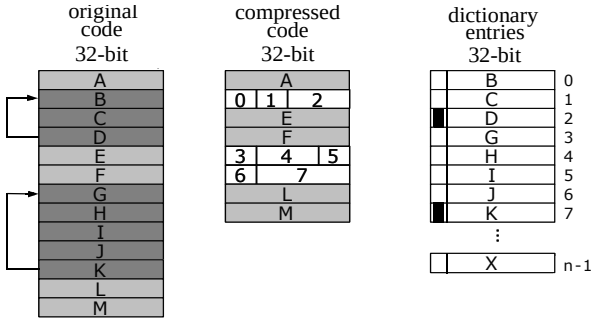


Figure 6: Dictionary code compression principle..

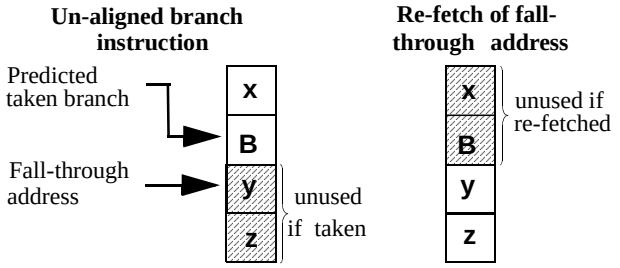


Figure 7: Excessively fetched bytes due to miss-prediction of an unaligned branch and re-fetch of the fall-through address.

wise would be brought into the processor completely unnecessarily. There is also reason to align the actual branch instructions. Many high-end embedded processors now employ some sort of branch prediction. Consider the case in Figure 7 where an unaligned branch in a fetch unit is wrongly predicted as taken. This means that not only are instructions y and z unused because of the taken branch, but also instructions x and B are unused when the fetch unit needs to be re-fetched because of the branch miss-prediction. Therefore, fall-through instructions should be considered as branch targets and pushed over to the next fetch-unit. Our solution is to have variable length branch instructions that can be elongated to fill up the rest of a fetch unit.

Branch compression: Given that 15-25% of all instructions in the MediaBench programs are branch/jump-instructions, it is important to be able to compress even these instructions. However, this is not as easy as for other instructions. A distinct bit-pattern that encodes a particular branch-instruction with a specific offset, might after compression of other instructions lead to be several totally different instructions since the branch distance for each branch most likely now is different. We have solved this by storing short branch offsets in the code word. On average more than 55% of the branch offsets can be fit in eight bits making branch instructions occupy in total 16 bits in the executable.

4.2 Code word architecture

The code word architecture is shown in Figure 8. It consists of three classes of code-words. The U-class is the uncompressed instruction set with the exception that bit 31 is always one denoting it as an uncompressed instruction. We have looked at the MIPS IV ISA (which is virtually identical to the PISA instruction set used in the SimpleScalar simulators) and found that it is indeed possible to reencode the instructions so that there is always a 1 in bit 31.

The G-class are code words for all possible instructions except relative branches. It is subdivided into three subclasses: G_1, G_2 and G_3. G_1 is the target code word for the most frequent instructions as it can be encoded in eight bits. Up to 125 G_1 code words are possible. For less frequent instructions the 16 bit G_2 code word can be used. As we now can fit an eight bit index for the dictionary, we can use up to 256 G_2 code words. Finally the G_3 has an additionally 8 bits for a total of 24 bits. The extra bits do not carry any information. Instead G_3 code words are used when padding is needed, for instance to align a branch target on the start of a fetch unit. By embedding this padding into the instruction code word, we do not need any extra decoding of padding instructions.

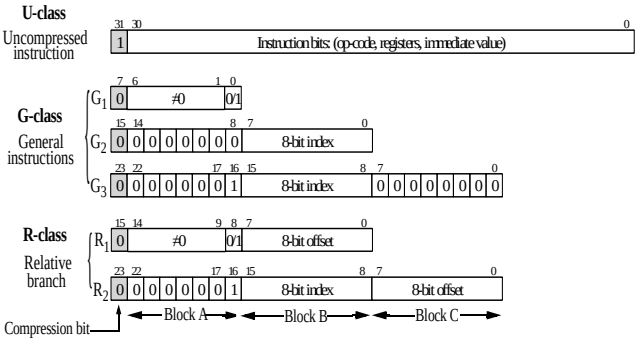


Figure 8: Code word bit-layout.

Finally, R_1 and R_2 are encodings of relative branches where the opcode and register argument combination are looked up in the dictionary while the offset is encoded with eight bits in the code word. This relieves us from the need to recalculate offsets in runtime. Again, the R_2 24-bit version is used for either more unusual branch instructions or when needed for alignment reasons. Relative branches which require more than 8-bit offsets can, unfortunately, not be encoded with short code words using our approach.

4.3 Code generation

Figure 9 shows the basic flow of the code compression process. Based on an instruction usage profile, as described in section 3, the compiler contains a compression engine which works on basic blocks, one at a time performing compression into code words as described previously and ending with an alignment so that each basic block always starts and ends at a word boundary. This means that a single instruction is never compressed in a basic block unless we thereby achieve a reduction of the number of fetch units in the basic block.

The compiler/linker also constructs the dictionary for use in the hardware. Later we will also consider the switching of dictionary contents as part of a context switch, but for now we are assuming that the contents is fixed. Given the code word architecture of Figure 8 we could potentially have five different dictionary tables for each different G- and R-class code words. However, in

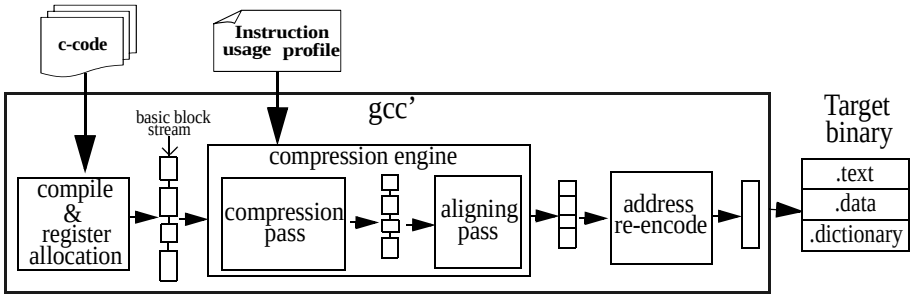


Figure 9. The code compression process.

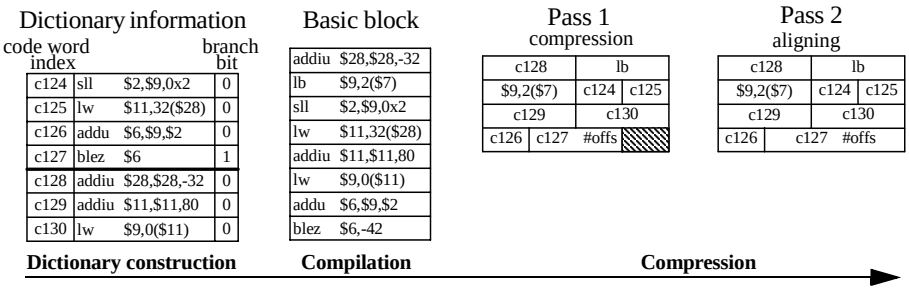


Figure 10: Compression engine, code compression and aligning

our implementation that we evaluate in section 6, we are using a 256-entry dictionary where instructions are basically stored in descending frequency from the profile.

Figure 10 shows a practical example of how the compression is done. It shows a part of the dictionary, as generated from the profile. We have here highlighted the portion between indexes that can be encoded with 7 and 8 bits, respectively. We also show a basic block to be compressed. The first addiu-instruction can be found in the dictionary with index c128. Since this index requires 8 bits, we encode this instruction with the G_2 format. The lb-instruction is not found in the dictionary and is therefore not compressed. It will stretch over two fetch units. The next two instructions are, however, found in the dictionary and can be encoded with G_1 code words. The following `lw` and `addiu` are, again, encoded with G_2 code words since their location in the dictionary need eight bits indexes. The addu-instruction is encoded with eight bits and the final branch instruction can also be indexed with eight bits making it possible to use the R_1 code word as the offset is within the limit. Since this compression results in an unaligned end of the basic block, the final branch-instruction is converted to an R_2 code word padding up the final byte.

We will now go over to show how this can be implemented in hardware before showing some actual results.

5. MICROARCHITECTURE

5.1 Reference architecture

The baseline architecture we are considering consists of a five-stage pipeline with in-order single-issue. This can be viewed as a moderately high-performance processor for embedded systems. There are separate instruction and data caches on-chip and the main memory off chip. There is no branch-delay slot and therefore it is important with branch prediction logic which also is appearing more and more in embedded processors. Table 2 shows the baseline architectural parameters of the processor.

5.2 Decompression stage

As mentioned before, the dictionary-based decompression engine is based on a modification of the fetch stage followed by a decompression stage between the fetch stage and the rest of the pipeline which stays unaltered. This is shown in Figure 1. The decompression stage adds an extra cycle to the branch miss-prediction penalty and therefore has a potential negative impact on performance in case the branch prediction performs poorly. We have made a detailed microarchitectural design taking all performance and power consequences into account, but it is out of scope for this paper to describe it here. The microarchitectural design can be found in [5] and in [17] we implemented the critical part of the microarchitecture of a previous feasibility study [7] verifying that the decompression logic and dictionary lookup would not affect the clock cycle time adversely.

5.3 Static versus dynamic dictionaries

The dictionary used here consists of a 32-bit wide table with 256 entries, as shown in Figure 11. About half of the table is dedicated to primarily 8 and 16 bits G_1- and R_1-class code words (see section 4.2), and the rest for G_2-, G_3- and R_2-class code words.

In most of our experiments of the compression architecture, we consider the dictionary contents to be static, i.e., it does not change during execution even though the executable changes. This is the reason for why we experiment with different aggregate profiles to allow for a limited multiprogrammed workload.

However, we are also considering the case when the dictionary works like a register-file and is thus possible to reload at context switches, if needed. The size of the dictionary is four times the size of the architected register-file which would significantly add to the context switch time. However, modern out-of-order multiple-issue super-scalar processors have physical register files that are of the

Table 2: Baseline architectural parameters.

Baseline processor:	
Issue width	1 (in-order)
INT/FP registers	32/32
Branch penalty	3 cycles for baseline
2-bit bimodal predictor	1024 entries
Branch target buffer (BTB)	128 entries, direct mapped
Return stack	8 entries
Memory system	
L1 I-cache	16kB, 2-way, 32 byte blocks, 1 cycle latency
L1 D-cache	16kB, 2-way, 32 byte blocks, 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

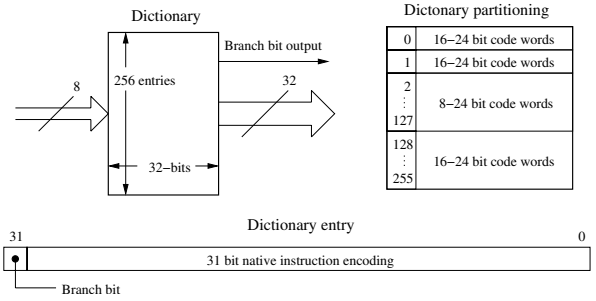


Figure 11: Logical view of the basic dictionary design, and organization.

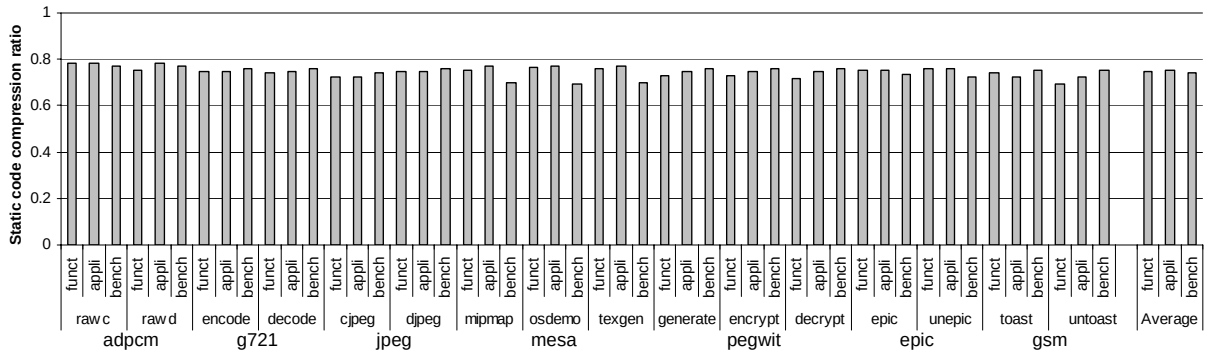


Figure 12: Static compression ratio for the different profiles.

same magnitude as the proposed dictionary, and in these cases the difference is not as big.

For the purpose of loading the dictionary with new information we propose a special load instruction: LoadDictionaryEntry *LDE*.

```
LDE entry, offset(rs)
```

The LDE instruction loads a 32-bit word from memory into the dictionary entry selected by the 8-bit immediate value denoted *entry*. To ensure simplicity and uniform usage of the intended context switch routine, the LDE instructions must not be compressed.

Furthermore, additional hardware modifications must be made to both dictionary design and data path. First of all, the memory elements constituting the dictionary entries now need to be volatile SRAM cells. To support writing of data into the dictionary, a 32-bit data-in port is added. In the processor pipeline, the WriteBack stage needs additional data and control paths to administrate the data to be written into the dictionary.

6. EXPERIMENTS

6.1 Experimental methodology

We have modified Wattch [4], which in turn is based on the sim-outorder simulator from SimpleScalar [6] to model our baseline architecture, and our code compression architecture described above. In addition, we have added energy models for the memory and bus interface taken from the IRAM project at Berkely [8].

Since there, for the moment, does not exist any compiler capable of generating a binaries using the proposed code word architecture, all executables retain their original format. Instead the binary is processed by an external post-compile compression tool that virtually compresses the instructions, generating an image of the compressed program organized as a translation table. The translation information is then used by the simulated pipeline front-end to model accesses made to the original uncompressed instruction memory space, as if compressed code actually was fetch from compressed memory space and decompressed in the pipeline.

6.2 Baseline experiments

Figures 12 to 15 show the baseline figures of the merits of our proposed dictionary compression scheme for the base architecture as described above. In Figure 12 we show the static compression ratio for the three levels of profiles as described in section 3. We achieve on average 75% compression meaning that we can save 25% of the main memory space for storing instructions. It is worth to notice that in many cases the bench-profile achieves the highest amount of static compression.

The dynamic compression ratio is shown in Figure 13. It is compared to the ideal case where the branch predictor is ideal resulting in fetching only the fetch units actually executed. In most cases the difference is small, but for some programs, such as rawc, the difference is, however, substantial. Overall, the dynamic compression ratio is close to our goal of 50% with an average of 50.4% when the functionality-specific profiles are used. It is easy to see, once again, that the bench-profile is useless. In several cases the instruction cache traffic is actually increased instead of reduced although the ideal case shows a reduction. This is because of miss-predicted branches. The programs raw{c|d}audio are the ones in with the highest branch-prediction miss rate resulting in worst dynamic compression rate for the bench-profile of all programs.

Figure 14 shows the breakdown of energy consumption for the fetch path and Figure 15 the same for the entire processor-memory system. The figures are normalized to the baseline architecture without instruction compression. We achieve on average 30-50% energy savings in the instruction fetch path which consists of the fetch stage and the branch prediction logic, accesses to the instruction cache and then memory accesses, if needed. On average it is 40%. Unfortunately, the instruction fetch path is just a fraction of the entire processor and in Figure 15 we can see that the total energy savings is around 20%. Here we have also added the extra energy consumption needed in the decompression stage.

It is expected that the performance of the dictionary compression processor is slightly worse than the baseline architecture because of the extra branch miss-prediction penalty cycle. Figure 16 indeed shows that this is the case. We have already established that the raw{c|d}audio programs have poor branch prediction performance and this also shows up as a 8-12% worse performance for the dictionary compression architecture as compared to the baseline architecture, even for the most specific profile. Although this is not very good, we will in section 6.3 show that for smaller cache sizes, we instead will have a performance advantage of using the code compression scheme.

6.3 Cache and line size explorations

Previous work that has showed impressive performance improvement have done so with artificially small instruction cache sizes between 128 and a few kbytes [13, 14, 15]. None of them have investigated the impact the cache line size has on the effectiveness of their schemes. In Figure 17 we investigate the normalized energy consumption with varying instruction cache size for the

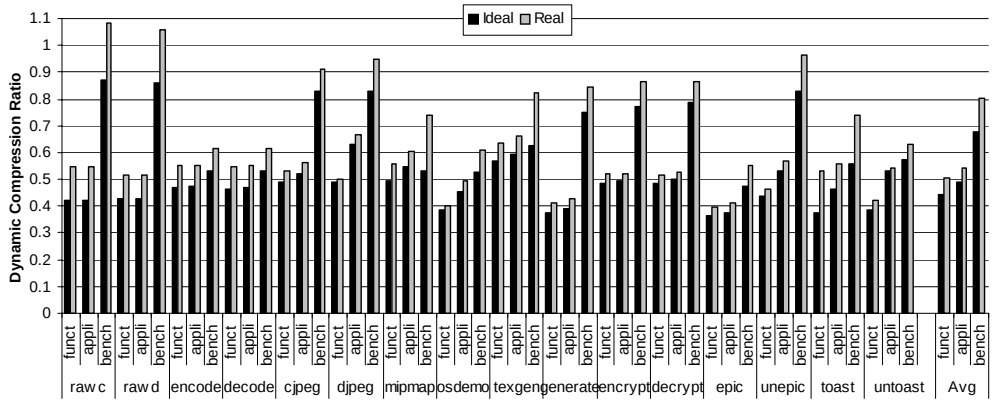


Figure 13: Dynamic compression ratio compared to the ideal case when no branches are miss-predicted.

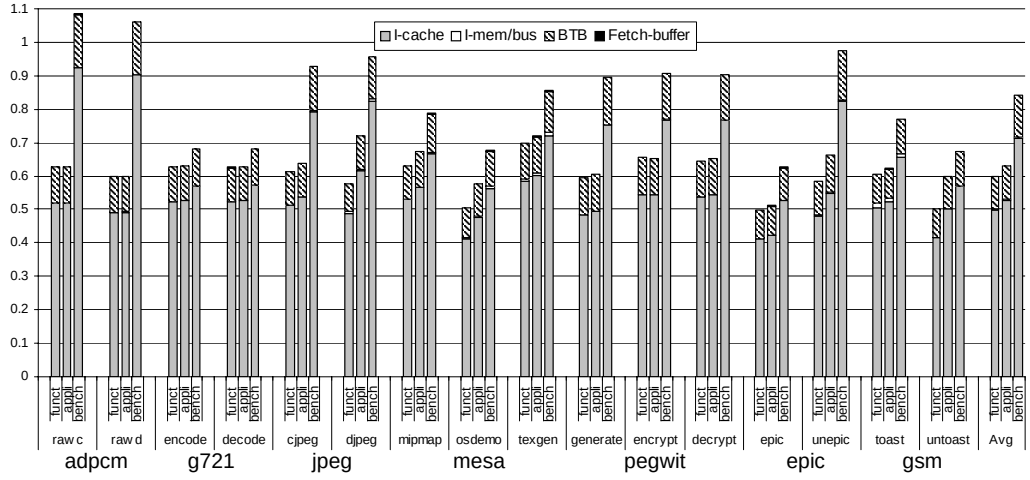


Figure 14: Energy ratio of the fetch path normalized to the baseline architecture.

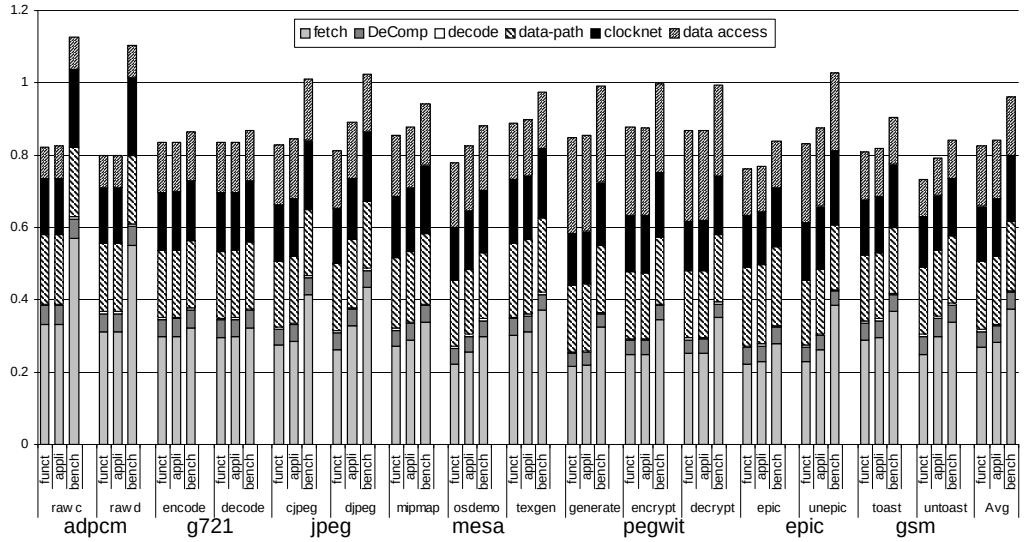


Figure 15: Energy ratio of entire processor and memory normalized to the baseline architecture.

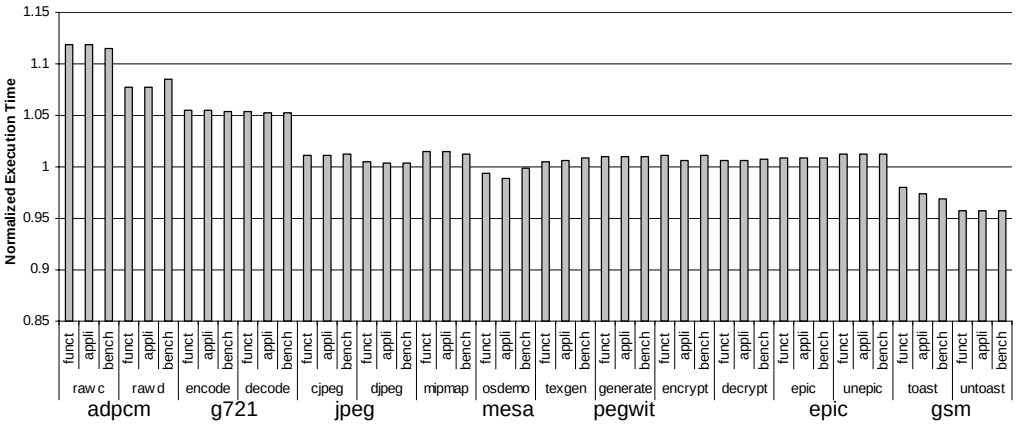


Figure 16: Execution time normalized to the baseline architecture.

g721application. All numbers are normalized to the baseline architecture with a 16 kB instruction cache. The g721 application is representative of the behavior of most MediaBench functionalities.

In Figure 17 (a) we plot the energy consumption of both the baseline architecture and the code compression architecture. The bars show the ratio between these curves. We can here see that an 8 kB instruction cache seems to be a sweet spot for this application, although the biggest improvement over the baseline architecture is achieved at a 4kB instruction cache. Figure 17 (b) shows the same for the code compression architecture but broken down in the individual components. We see that with smaller instruction caches we get a reduction in energy consumption in the cache to some point when word line access power dominates and we no longer achieve any energy gain from the instruction cache. In addition, the bus and memory energy starts to become noticable. Previous work has neglected the increased energy consumption of the main memory with very small instruction caches.

In Figure 18 we see how the small caches sizes impact the performance. While we have a slight performance disadvantage for big instruction caches due to the extra pipeline stage, it is clearly so that with smaller caches we improve the performance over the uncompressed instruction set significantly. In fact, for this application, when we reduce the cache size by half, we get about the same performance, or better, as the uncompressed architecture with unchanged instruction cache. It is clear that for embedded systems where memory size is crucial, a post-cache decompression scheme such as the one we present here is to prefer over a pre-cache scheme that decompresses instructions into the cache.

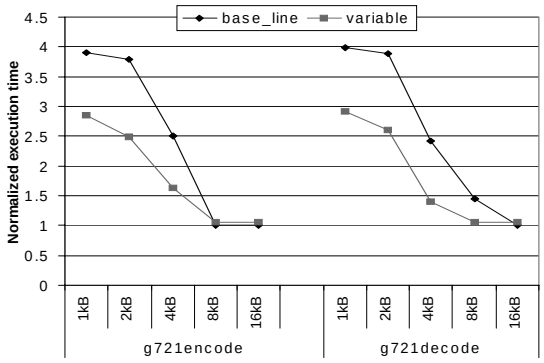


Figure 18: Normalized execution time for the g721 application for different cache sizes.

Finally, Figure 19 investigates the effect different cache line size have on performance and energy consumption for the same application as above: g721 encode and decode. In this case, a 2 kB instruction cache is used and the line size is varied from 8 bytes up to 128 bytes. As expected, we get a trade-off between large and small lines. In a large line we can utilize the spatial locality of code accesses greatly and that improves performance. On the other hand, longer cache lines are more costly to read from in terms of energy which leads to a slight increase in energy consumption when 128 bytes are used over 64 byte lines. However, small cache lines are also not good for energy consumptions as they do not utilize the locality and the memory needs to be consulted more often.

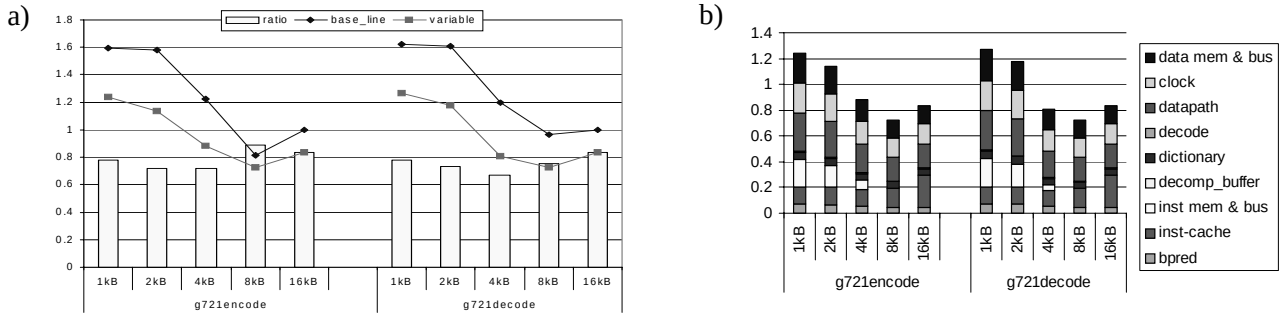


Figure 17: Normalized energy consumption for the g721 application for different cache sizes.

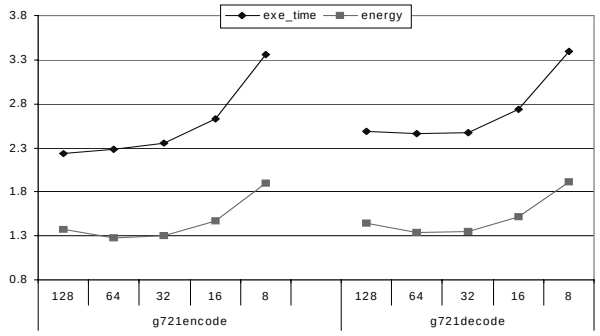


Figure 19: Normalized execution times and energy consumption for the g721 application for different cache line sizes given a 2 kB instruction cache.

6.4 Impact of dynamic dictionaries

Previous work on dictionary code compression have always considered the dictionary to be static and never looked at multiprogrammed workloads, which are becoming more and more common also in embedded systems. We present a novel idea that the dictionary contents is part of a process' context and reloaded each time there is a context switch. Of course, if needed, several processes/ threads could share the same dictionary context, reducing the number of times the dictionary needs to be reloaded. The idea is to investigate the trade-off in performance and energy of using as specific dictionary contents as possible for each program and change dictionary contents when a context switch occurs. We compare the results to the case of using an aggregated profile to determine the dictionary contents.

Table 3: Number switches as function of execution & operating system period time.

Application	Application Execution time	OS_period time		
		10 ms	50 ms	100 ms
adpcm	43.8 ms	6	2	2
g721	2.35 s	237	49	25
jpeg	75.7 ms	9	3	2
mesa	909 ms	92	20	11
pegwit	522 ms	54	12	7
epic	239 ms	25	6	4
gsm	962 ms	98	21	11
all	5.11 s	512	104	53

We model the extra time and energy to reload the dictionary assuming an extra 256 LDE-instructions that always miss in the cache. With this assumption, the time needed to load the dictionary is ~34000 cycles or 85 µs. In addition of modelling the energy and time for retrieving dictionary contents from the memory, we have also modelled the extra energy a write port to the dictionary memory entails. The application execution times and the number of context-switches performed during execution of each application using period times between context switches is shown in Table 3. The number of required switches is calculated using equation 2.

$$\text{switches} = \left\lceil \frac{\text{application\_exe\_time}}{\text{OS\_period}} \right\rceil + 1 \qquad \text{(eq. 2)}$$

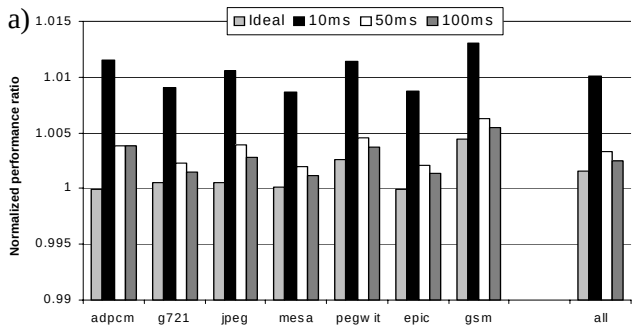
The base case used in this experiment is the case for when all applications are simulated using dictionary contents based on the appli-profile, which is an aggregated profile of the functionalities within the application. For the case denoted *all*, which includes all the 16 individual test programs, the bench-profile was used. Performance and energy measurement for all these base cases were obtained using a fixed contents dictionary.

To further visualize the extent of the extra overhead, an additional comparison point is added to the graphs, denoted *ideal*, which implies that loading the dictionary with each programs' corresponding funct-profile comes without any cost. All results, both performance and energy ratio for the different applications are shown in Figure 20, normalized to the above described base-case.

Figure 20 (a) shows the results regarding performance ratio and the imposed overhead in execution time for loading new dictionary contents. Note the magnified scale on the y-axis. It is obvious that the overhead of reloading the dictionary is negligible with a 1% increase in execution time in the worst case.

Figure 20 (b) shows the results regarding energy consumption. Although we can achieve energy savings of up to 4% as compared to the case with aggregated profiles, some applications, such as adpcm and pegwit, actually exhibit a worse energy consumption using context switching of dictionary contents based on specific profiles. The overhead of making the context switch is not worthwhile considering that the functionalities in these applications are very similar when it comes to their specific profiles. However, when we compare to the case where we use the bench-profile ("all" in Figure 20), the energy savings is considerate.

In a real system, the mix of programs determines whether dynamic reloading of dictionary contents should be done at context switch or not. For programs that share similar profiles, time and



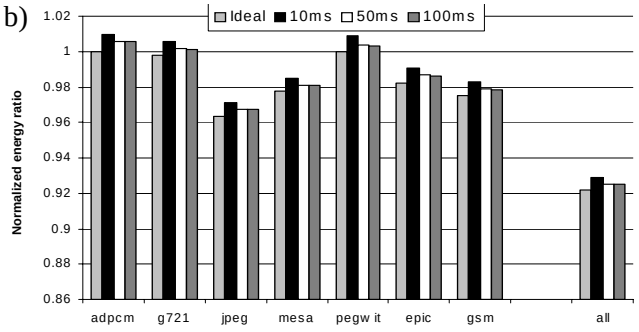


Figure 20: a; Performance and energy ratio when taking the loading of dictionary during task switches into account for the seven applications and for a work load consisting of all applications.

energy can be saved by not reloading the dictionary, although the overall overhead is anyway negligible.

7. CONCLUSION

To summarize, we have presented a code compression scheme, based on off-line instruction profiling to determine the dictionary contents used in the decompression of instructions that takes place in a new pipeline stage directly after the fetch stage in an ordinary five-stage pipelined RISC-processor. A novel aspect of the presented scheme is the way we can produce aggregated profiles so that a combination of programs can make use of the dictionary code compression. Also we show the effects of cache size and line size on energy consumption and performance which never has been done before. In addition, we have a detailed microarchitectural model, reported elsewhere, outlining all potential performance problems in detail, all modelled in our simulation model. In particular, previous work have not satisfactorily described problems that occur with branch instructions, in particular when branch prediction schemes are taken into account. Finally we are first in showing that the dictionary contents very well can be part of a process' context with negligible performance overhead and substantial energy savings. With smaller cache sizes, the performance overhead will be converted into a significant performance gain of using dynamic dictionaries.

Our scheme achieves a good static and dynamic compression when functionality-specific profiles are used to determine the dictionary contents. This results in fetch-path energy savings of up to 50%. Our scheme can be augmented with multiple dictionaries for different code word classes, although this will increase the overhead of reloading the dictionary on context switches. However, by using different aggregation on different tables, this should be possible to reduce. One might also consider having different dictionaries for library code and the source code. Different dictionaries for operating system code and the user-level code etc.

We also believe that the scheme is possible to extend to superscalar processors with even more performance and energy gains as with one fetch from the instruction cache memory, we get in average two instructions that could potentially be issued in parallel.

Acknowledgment: The research in this paper has been supported by the HiPEAC European Network of Excellence.

REFERENCES

[1] G. Albera and R. I. Bahar, Power and Performance Tradeoffs using Various Caching Strategies, in *Proceedings of the International Symposium on Low Power Electronics and Design*, Monterey, CA, August 1998, pp. 64-69.

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

[3] E. Berg and E. Hagersten, Fast Data-Locality Profiling of Native Execution, in Proceedings of ACM SIGMETRICS'05, Banff, Canada, June 2005, pp. 169-180.

[4] 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, ISCA'00*, June 2000, pp. 83-94.

[5] M. Brorsson and M. Collin, *A Microarchitecture for Profile-Based Code Compression using Code Word Dictionaries*, Technical report, Dept. of Electronic, Computer and Software systems, Royal Institute of Technology, May 2006. Submitted for publication.

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

[7] M. Collin and M. Brorsson. "Low Power Instruction Fetch using Profiled Variable Length Instructions", in *Proceedings of the IEEE International SoC Conference*, Sept. 17-20, Portland, Oregon, 2003.

[8] 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.

[9] M. R. Guthaus, J. S. Ringenberg, D Ernst, T. Austin, T. Mudge, and R. Brown, MiBench: A free, commercially representative embedded benchmark suite *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[10] K. D. Kissell. MIPS16: High-density MIPS for the Embedded Market, in *Proceedings of Real Time Systems'97* (RTS97), 1997.

[11] 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.

[12] 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.

[13] H. Lekatsas, J. Henkel, and V. Jakkula, Design of an One-cycle Deompression Hardware for Performance Increase in Embessded Systems, in *Proceedings of the Design Automation Conference DAC2002* (June 2002), pp. 34-39.

[14] H. Lekatsas, J. Henkel, and W. Wolf, Code Compression for Embedded System Design, in *Proceedings of the 37th Design Automation Conference*, June 2000, pp. 516-521.

[15] E.W. Netto, R. Azevedo, P. Centoducatte, and G Araujo, Multi-Profile Based Code Compression in *Proceedings of the 41st Design Automation Conference*, June 7-11, 2004 Page(s):244 - 249.

[16] R. Phelan, *Improving ARM Code Density and Performance*, ARM Ltd white paper, June 2003.

[17] Sleeba B., Collin M., and Brorsson M. *An ASIC implementation and evaluation of a profiled low-energy instruction set architecture extension*. Technical report, KTH Microelectronics and Information Technology, Oct 2003. http://web.it.kth.se/~matsbror/papers/sleeba_variable_asic.pdf

[18] C. L. Su, C. Y. Tsui, and A. M. Despain, *Saving power in the control path of embedded processors*, IEEE Design Test Comput., vol. 11, no. 4, pp. 24--30, 1994.

[19] J. L. Turley, *Thumb Squeezes ARM code size*, Microprocessor Report, 9(4), pp. 1-5, 27 March 1995.

[20] N. Vijaykrishnan et al., Energy-driven Integrated Hardware-Software Optimizations using SimplePower, in *Proceedings of the 27th Annual International Symposium on Computer Architecture, ISCA'00*, June 2000, pp. 95-106.

[21] A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture", in *Proceedings of MICRO´25*, December 1992, pp.81-91.