Design Automation for Embedded Systems (2024) 28:245–274
https://doi.org/10.1007/s10617-024-09290-2

Energy-efficient instruction compression with programmable dictionaries

Joonas Multanen[1] · Barry de Bruin[1] · Henk Corporaal[2] · Pekka Jääskeläinen[1]

Received: 9 May 2024 / Accepted: 1 November 2024 / Published online: 16 November 2024

Abstract
To improve the energy efficiency of computation, accelerators trade off performance and energy consumption for flexibility. Fixed-function accelerators reach high energy efficiency, but are inflexible. Adding programmability via an instruction set architecture (ISA) incurs an energy consumption overhead, as instructions are fetched and decoded. To reduce it, hardware-controlled instruction caches and software-controlled components, such as loop buffers and (programmable) dictionaries improve the energy efficiency of instruction streams in embedded processors. Reducing the instruction overhead with code compression is well established and dictionary compression has been an effective approach due to its simplicity. Compared to static dictionaries, adding programmability improves the effectiveness. However, run-time-programmable dictionary compression and its effect on energy consumption has not been thoroughly studied. We describe a scheme to target energy efficiency by using fine-grained programmable dictionaries in embedded compute devices. Guided by compile-time analysis, the dictionary contents are changed during execution. On CHStone and Embench suites, our method reduces energy consumption on average by 11.4% and 3.8% with negligible run-time overhead. The addition of a loop buffer further reduces the energy consumption by 19.8% and 4.5% in the two suites. Our results indicate that programmable dictionary compression allows further energy reductions over an already highly tuned instruction stream.

Keywords Code compression · Computer architecture · Embedded systems · Energy efficiency · RISC-V · VLIW

✉ Joonas Multanen
joonas.multanen@tuni.fi

Barry de Bruin
egbart.debruin@tuni.fi

Henk Corporaal
h.corporaal@tue.nl

Pekka Jääskeläinen
pekka.jaaskelainen@tuni.fi

[1] Tampere University, Tampere, Finland

[2] Eindhoven University of Technology, Eindhoven, The Netherlands

1 Introduction

While dedicated fixed-function accelerators offer the best energy efficiency, they are inflexible and do not allow executing tasks that they were originally not designed for [1]. By adding programmability to devices, their flexibility can be improved with the drawback of increased area and energy consumption caused by the instruction stream from memories to compute elements. As this does not directly contribute to the processing of data and can consume up to 70% of total system energy budget [2–4], minimizing it is appealing. As an example: Even a highly optimized binary precision deep neural network accelerator [5] spends up to 11% of its energy budget on the instruction stream. This suggests that even though the topic has seen little research efforts in the recent years, there is room for improvement. Moreover, the massive popularity of *artificial intelligence* (AI) applications in the last years has resulted in the re-emergence of *very long instruction word* (VLIW) architecture processors, as they have been implemented in recent accelerator designs such as the Intel Gaudi [6] and AMD Versal [7]. While they move the complexity of dynamic multi-issue hardware to the compiler, the effectiveness of VLIW architectures is dependent on the instruction level parallelism that can be extracted from the target workloads, and can result in excessive overheads in the instruction stream.

Instruction streams in programmable processors are typically built in a hierarchical manner, as accessing large memory components that are located relatively far from processing is expensive compared to those that are small and located close to it. In this article, we focus on the first level (L0) of instruction memory hierarchy, as it is accessed the most frequently. Typical components used here include filter caches [8] and *loop buffers* (LB) [9], which both aim at reducing the cost of executing relatively small loops. Although loop buffers result in energy-efficient loop execution [10] due to their small access energy, removing accesses to the more costly memory hierarchy levels when they are in use, and removing the loop iteration overhead [11], they do not reduce energy consumption outside of loops. In addition, they are typically limited to straightforward loops with no complex control structures.

Another technique that reduces the amount of accesses to higher memory hierarchy levels, is code compression. In early publications [12], it was proposed as an approach to reduce code size in compute systems and alleviate the area and economical cost of memory in systems. In addition, it allowed a reduction in memory bandwidth requirements. Although the majority of the previous work [12–22] measures the effectiveness of code compression with *static compression ratio* (CR), there have been proposals [19–25] to target *dynamic CR*, which measures the amount of instruction bits fetched from external memory. Since components lower in the memory hierarchy consume less energy, and improving dynamic CR reduces the number of accesses to higher hierarchy levels, this translates to less instruction stream energy consumed. As execution amounts of different instructions in programs are biased, i.e. depend on the input data, compression should emphasize the most executed instructions in order to reduce the amount of bits fetched, improving the energy efficiency of the instruction stream.

Although statistical compression methods such as Huffman compression reach better CRs than dictionary-based approaches, they result in performance overheads [16] due to the sequential decompression required for the variable-length codewords, resulting in variable latency. Thus, research efforts have been heavily concentrated on variants of dictionary compression [13, 15, 18, 20–26]. Although the simplest dictionary compression scheme involves placing instructions into a dictionary and pointing to them with indices in program code, effective implementation requires design choices on whether to compress all instructions or

not, and if instructions should be divided into parallel dictionaries as this typically reduces the total number of dictionary entries required. Moreover, run-time-programmable dictionary compression schemes, where dictionary contents can be changed during execution, have been less studied. In these schemes, the granularity of dictionary programming requires an additional design decision.

The contributions of this work are as follows:

- A code compression scheme for embedded systems to increase system energy efficiency by using run-time programmable dictionaries and a low-overhead decompression hardware design,
- a method to statically analyze *control flow graphs* (CFGs) and to organize program basic blocks into *code compression regions*, and a heuristic to form instruction bundles from the most profitable instructions,
- an analytical model for estimating instruction stream energy consumption,
- methodology for sizing the dictionaries and designing instruction streams incorporating them, and
- evaluation on a single-issue and a wide-issue machine to demonstrate applicability to different architectures.

This article builds upon our previous work [27], where we introduced the code compression scheme. More precisely, we described a run-time programmable dictionary compression scheme accompanied by two algorithms: One to group program *basic blocks* (BBs) into regions suitable for code compression, and the second to selectively choose suitable bundles of instructions to compress with the help of an *occurrence-cost* metric. We showed that selectively choosing instructions for compression outperforms the previous state-of-the-art fine-grained programmable dictionary compression method [25], where all program instructions are compressed.

In addition to entailing the above work, this article provides the following additional contributions: 1) It demonstrates that our approach can be applied effectively to a long instruction word architecture in addition to a previously evaluated single-issue RISC-V architecture, 2) provides an analytical model for estimating instruction stream energy consumption, 3) adds methodology for instruction stream design and dictionary sizing with the help of the analytical model, 4) evaluates the interaction between a loop buffer and the proposed code compression, and 5) adds benchmarks from the Embench suite in addition to the CHStone suite. Our results show that designing and implementing our method in a system containing an already energy-efficient loop buffer further reduces energy consumption in applications with complex control flow.

The rest of this paper is organized as follows. Section 2 reviews the previous work on instruction compression. Section 3 discusses energy-efficient design choices related to code compression. Section 4 continues by describing the proposed compression approach. Section 5 introduces the analytical model used in instruction stream hierarchy design space exploration, and Sect. 6 provides an evaluation of the proposed method and presents results. Finally, Sect. 7 describes the conclusions.

2 Related work

In this section, we review different types of previously proposed instruction compression methods with an emphasis on dictionary-based approaches. We then revise run-time

programmable dictionary compression methods from the perspective of programming granularity. Lastly, we examine more closely the methods used to choose entries in *selective* dictionary compression schemes, where all program instructions are not compressed.

2.1 Instruction compression approaches

Previous work has proposed a variety of approaches for instruction compression. Ideally, decompressing the fetched instructions should result in minimal overheads in energy consumption, area and latency. Statistical compression methods assign variable-length codewords based on the distribution of symbols. While they result in impressive compression ratios, they require variable-latency decompression. On the other hand, dictionary-based compression allows predictable decompression, but does not reach as good compression ratios.

Wolfe and Chanin [12] used Huffman code to compress instructions. Instructions were fetched into caches in their compressed form and decoded for execution. Another statistical method was proposed by Lin et al. [17], who used *Lempel-Ziv-Welch* (LZW) compression on program basic blocks. In this approach, a coding table is generated dynamically by the hardware during execution in both compression and decompression. Upon encountering a branch target, the table is cleared.

Conte et al. [28] studied a code compression scheme for VLIW architectures. In this scheme, only *no-operations* (NOPs) were compressed. The maximum size of the compressed instructions, *MultiOps*, was the number of function units in the VLIW processor. contained and the evaluations were carried out on a *TINKER* VLIW testbed.

Lefurgy et al. [13] observed that programs contain only a small amount of instructions that are never reused. Compressing these instructions would not have been beneficial but instead resulted in an overhead. Thus, the authors *selectively* compressed instructions so that uncompressed and compressed instructions were interleaved in program code. The authors evaluated various input sizes for compression ranging from patterns inside instructions to sequences of consecutive instructions. They conclude that the best static CR is obtained when instruction patterns are allowed in the compression. Our proposed method similarly compresses instructions selectively and interleaves them with uncompressed instructions.

Similarly to Lefurgy et al., Benini et al. [23] studied four variants of selective instruction compression. The authors interleaved dictionary-compressed 8-bit and uncompressed 32-bit instructions in memory. Each variant used a *mark* of 8 bits to indicate an uncompressed instruction after it. Although the authors aim to reduce energy consumption, the effect on it is not directly evaluated. Instead, the amount of memory accesses and memory bus utilization are reported. Moreover, overheads stemming from the decompressor implementation are not evaluated.

Compared to full instructions, the individual instruction fields contain more similar entries when observing between instructions, requiring less bits to represent the different instructions. Thus, previous work has looked at how to split instructions into parallel dictionaries, that are accessed simultaneously when decompressing. Lefurgy et al. [29] studied the effect of CodePack compression found in the IBM PowerPC processor, where 32-bit instructions are partitioned into two 16-bit parts and compressed separately with parallel dictionaries. Ishiura and Yamaguchi [30] proposed an algorithm to partition instructions in VLIW architectures. Fields of the instructions to compress were constructed in parallel by iteratively merging them. Our proposed method adopts a similar approach as these works so that instructions are partitioned into fields and compressed with parallel dictionaries.

Table 1 Comparison of code compression methods

	Type	Update granularity	Compr. target	Decomp. on fetch	Selective	Year
CCRP [12]	Huffman	Program	Static CR	No	Yes	1992
Lefurgy et al. [13]	Dict.	Program	Static CR	Yes	Yes	1997
Benini et al. [23]	Dict.	Fixed	Dyn. CR	Yes	Yes	1999
Lekatsas et al. [18]	Dict.	Program	Perf. + static CR	Yes	Yes	2002
Benini et al. [19]	Mixed	Program	Static/ dyn. CR	Yes	Yes	2004
Netto et al. [20]	Dict.	Fixed	Static + Dyn. CR	No	No	2004
Das et al. [21]	Dict.	Program	Static + Dyn. CR	Yes	No	2005
Brorsson & Collin [24]	Dict.	Context Switch	Dyn. CR + Energy	Yes	Yes	2006
Xie et al. [14]	V2F	Fixed	Static CR	Yes	No	2006
Lin et al. [17]	Table based & LZW	Branch block	Static CR	Yes	Yes	2007
Seong et al. [15]	Dict. + Bitmask	Program	Static CR	Yes	Yes	2008
Thuresson et al. [25]	Dict.	Basic block	Dyn. CR	Yes	No	2009
Bonny & Henkel [16]	Huffman	Fixed	Static CR	Yes	No	2010
Shriv. & Mishra [22]	Dict.	Program	Dyn. + Static CR	Yes	Yes	2011
This work	Dict.	Basic block	Dyn. CR + Energy	Yes	Yes	2024

Seong and Mishra [15] proposed to improve code compression with the use of an orthogonal method. That is, this approach can be implemented alongside code compression. In addition to dictionary compression, the authors used a technique, where the difference between two instructions was encoded as a *bitmask* in conjunction with the dictionary index. During decompression, instructions were obtained by performing an XOR operation between the bitmask and the dictionary entry pointed by the index. This allowed reusing dictionary entries between instructions, improving the compression ratio. We do not implement orthogonal methods such as this as they are out of the scope of this publication.

Das et al. [21] applied dictionary compression to a variable-length *instruction set architecture* (ISA). The authors used trap instructions to indicate compressed codewords. The authors argued that even though variable-length ISAs are already compressed so that short codewords are used for the most common instructions, there is still room for additional compression using a dictionary-based approach. In this article the two evaluated processor architectures feature fixed-length instructions.

Lekatsas et al. [18, 31] evaluated performance improvements achieved with code compression. The authors concluded, that compression was able to increase cache hit ratios, resulting in improved performance.

Netto et al. [20] used four different codeword lengths to index their dictionary. Each codeword was preceded by a field indicating its length. The authors then assigned instructions to dictionaries based on their occurrence frequency. In the proposed method, only a single codeword length is used for compressed instructions.

Bonny and Henkel [16] used Huffman compression in conjunction with instruction *splitting*, where full instructions were divided into variable-length parts to allow better Huffman compression. In addition, the authors used instruction *re-encoding*, where they identified *don't care* bits in instructions and used them to improve compression efficiency by matching the bits with other instructions.

Xie et al. [14] argued that the variable-latency decompression required for statistical compression algorithms could be addressed by using a *variable-to-fixed* (V2F) compression scheme. The authors proposed to use a Tunstall coding tree for compression. This approach allowed parallel decompression of multiple codewords when using a statistical compression algorithm.

2.2 Run-time programmability

The majority of previous research has proposed to either have fixed decompression hardware [14, 16, 20, 23] or program dictionaries once at the start of program execution [12, 13, 15, 19, 21, 22]. However, programs may exhibit phases during which the instruction mix changes, and a single set of dictionary entries may not represent the different phases effectively. In this light, adding run-time programmability to dictionary compression is appealing, although it results in overheads.

There have been relatively few proposals, where the dictionary contents can be dynamically programmed during execution. Brorsson and Collin [24] evaluated a compression scheme, where the dictionary contents could be programmed during context switches in processors that support multithreading. The authors also evaluated sharing dictionary contents between multiple contexts. They concluded, that energy consumption and dynamic CR were improved by programming dictionaries at context switches, if the instruction mixes between contexts were different. Moreover, the authors stated that the overhead of programming all entries of a 256-entry dictionary at context switches was negligible.

Thuresson et al. [25] proposed to update dictionaries at the granularity of program BBs. In their approach, entries were programmed individually by using special instructions. In addition to the fine-grained dictionary programming, the authors partitioned instruction words into parts which were compressed individually by using parallel dictionaries.

2.3 Criteria for selective compression

Ideally, all the instructions of a program should be compressed. However, in real programs compressing some instructions offer greater benefits than others. This is due to the varied occurrence and execution counts of different instructions. Also, compressing all instructions may not lead to the best static or dynamic CR. As more entries are added to a dictionary, the number of bits required to index them increases. In this light, previous work has selectively compressed instructions. Li and Chakrabarty [32] showed, that selecting optimal entries from program instructions for dictionary compression is an NP-hard problem. In the following, methods proposed by previous work to select dictionary entries are reviewed.

While previous work on selective compression has mostly concentrated on either targeting static or dynamic CR, Shrivastava and Mishra [22] proposed to gain dual benefits by targeting both. The authors first used compression to minimize dynamic CR with the guidance of a dynamic profile, and then compressed the result again with the target of static CR.

Lin et al. [17] aimed to increase the size of individual code regions to compress. The authors identified *branch blocks*, also known as superblocks [33], from CFGs. These are regions of BBs, where branching is only allowed into the first BB. This prevented alignment issues when branching into compressed code.

Benini et al. [23] used dynamic profiling to select the most used instructions as entries into a dictionary. The authors concluded that 256 entries were sufficient to cover the majority of instructions in their benchmark set.

Thuresson et al. [25] proposed to compress all instructions in a program, allowing fixed-length instructions. This was combined with their scheme where individual dictionary entries could be programmed during execution. To reuse entries already residing in the dictionary, the authors proposed a recursive algorithm to decide the location of dictionary programming instructions. At the beginning of each BB, instructions were programmed into dictionaries if they were not already in the dictionary. For each entry, the BB's predecessors were recursively examined, and if they contained the same entry, the programming instruction was pushed into that predecessor. In case of a BB requiring more entries than fit into the dictionary, the BB was split. Immediate values were not compressed, as they required fixing after compression in case they were used as branch target addresses.

Rawlins and Gordon-Ross [34] studied the interplay of code compression, loop caching and L1 cache tuning. The authors concluded, that cache tuning alone produced most of the energy savings. Adding a loop buffer allowed to further reduce energy consumption. The authors also evaluated a scheme where Huffman-compressed instructions were stored in an L1 instruction cache and decompressed to the loop buffer. The conclusion was that a loop buffer increased performance and reduced energy consumption, as it allowed removing the decompression overhead when executing loops. The authors also concluded, that an algorithm with a lower decompression overhead compared to Huffman should be used. Although this work did not directly produce novelty for code compression, it is important from the perspective of energy consumption when designing systems containing loop buffers and code compression. Our work evaluates a similar system setup, but replaces Huffman compression with our proposed code compression which has a predictable decompression overhead.

Table 1 presents a qualitative summary of the previous work. The majority of the reviewed previous work uses dictionary compression, with some works utilizing statistical compression algorithms. Most works have used static CR as the main metric for evaluating code compression, with some works using dynamic CR or energy consumption. Most works have also proposed to decompress instructions when they are fetched. That is, decompression is performed adjacent to instruction decoding. Most works have also proposed to use selective compression of instructions.

3 Design choices for energy-efficient code compression

Previous work on code compression has mostly concentrated on optimizing static CR, which allows the memory footprint of programs to be reduced. On the other hand, targeting dynamic CR and reducing the overall energy consumption is a less studied direction. With the target of energy efficiency, the dictionaries should be designed as small as possible, but still allow a good dynamic CR. With this in mind, the following discusses compression design choices with the goal of reducing energy consumption.

In the rest of this section, we consider approaches suitable for dictionary compression. Although methods that produce variable-length code words such as the statistical Huffman compression [35] or heads-and-tails [36] have been used for instruction compression, their decompression incurs a performance overhead due to the variable latency [16, 34, 37].

3.1 Selective instruction compression and programmability

Ideally, all the instructions in programs should be compressed to minimize static or dynamic compression ratio. However, in dictionary compression this leads to large dictionaries if the application instruction mix is heterogeneous, as even seldom occurring instructions have to be stored in the dictionary. Previous work [23] has noted that the majority of program instructions can be represented using a limited amount of dictionary entries that are fixed for the duration of the program. That is, the dictionary sizes can be kept reasonable by selectively compressing instructions.

However, having a static dictionary for the program duration is inflexible. Another point of consideration is, that instruction mixes typically vary between programs, program phases, and even loops in the program phases. In this sense, dictionary programmability [24, 25] allows changing the dictionary entries during run-time. By compressing all instructions and programming the missing entries into dictionaries at the start of each basic block, Thuresson et al. [25] achieved impressive dynamic CR. However, this results in an excessive amount of dictionary programming at run-time, when all the entries required for a loop do not fit into the dictionary. While this is not an issue in cases where the dictionaries can be tailored to a set of target applications that contain simple loops, it can result in excessive performance overhead in tasks not known at design time. This overhead reflects negatively on energy efficiency.

In this light, if the programmability is combined with selective compression, dictionary programming inside loops can be avoided, as we show later in this article.

3.2 Bundling

By limiting the amount of variable instruction lengths, the decompression overhead can be reduced with the trade-off of compression ratio. In straightforward selective dictionary com-

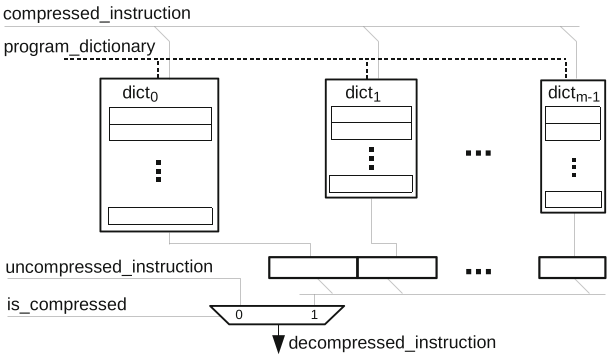


Fig. 1 Decompressor with m parallel dictionaries that each have individual number of entries and are programmable. Compressed instructions consist of indices pointing to the dictionary entries

pression, the number of variable lengths is two: One size for pointing dictionary indices and another for uncompressed instructions. However, if the compressed and uncompressed instructions are allowed to be placed arbitrarily in memory, the number of combinations how the instructions can be aligned is higher than two, when the uncompressed length is not divisible by the compressed length. To solve this problem, consecutive compressed instructions can be arranged as *bundles* and aligned with uncompressed instructions, so that a fetch word always contains either a bundle, or an uncompressed instruction [23]. The compressed bundles are different, but resemble VLIW instruction bundles, which consist of a fixed number of operations grouped together.

3.3 Dictionary partitioning and sizing

Partitioning instructions into fields and compressing them using parallel dictionaries has been shown to improve the static CR [30]. Compared to a single large dictionary, this allows the individual dictionaries to represent more instructions with relatively fewer entries. This is due to the amount of different entries varying between fields. Typically register and immediate field values experience the most changes during program execution. Implementing individual dictionaries for these fields allows good compression ratios, whereas fields with lower rate of change can be combined and placed into the same dictionary. Dictionary partitioning is illustrated in Fig. 1, where the amount of entries in dictionaries can be sized individually based on target applications or the ISA characteristics corresponding to the field. Bypassing the dictionaries and directly using uncompressed instructions is done using the multiplexer in the lower region of the figure.

Deciding the number of parallel dictionaries and instruction bit indices that will be compressed in each dictionary requires careful attention. Assuming that the instruction fields between different instructions contain repetition, selecting a single dictionary results in poor CR, as the number of instruction combinations is high in a flexible instruction set processor. On the other hand, dividing instructions excessively into many dictionaries also results in poor CR, as each dictionary requires individual addressing bits in the compressed instruction. Moreover, some instruction fields use more encodings than others depending on their function. This requires consideration for the amount of entries in each parallel dictionary. The optimal amount of dictionaries and the number of their entries depends on the target applications and their instruction mixes.

3.4 Entry selection

In addition to selectively compressing instructions and aligning compressed bundles with uncompressed instructions, program code structures should be taken into account to minimize dynamic CR and energy consumption. In schemes with programmable dictionaries, since there is an energy (and performance) overhead from the dictionary programming, cost analysis is required per code region to determine if it should be compressed.

Moreover, partitioning instructions into fields and using parallel dictionaries in a scheme with selective compression complicates the entry selection. Let us assume that the dictionary sizes should be minimized to optimize energy consumption. As compressing an instruction requires entries from all the parallel dictionaries, and the occurrence of entries varies between instructions, it is not trivial to select entries resulting in minimal dynamic CR. Moreover, when the selectively compressed instructions are grouped as fixed-size bundles, the cost of the whole bundle has to be considered.

4 Programmable dictionary code compression for energy-efficiency

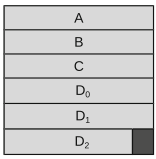
This section introduces our method which builds upon the energy-efficient design choices discussed in Sect. 3. It combines run-time programmable, individually sized parallel dictionaries with selective instruction compression, as well as compressing instructions in fixed-size bundles in order to avoid costly variable-length decoding. Details of the hardware implementation were introduced in [27], and as they are relatively straightforward, we do not further elaborate them in this article.

Our method primarily targets the reduction of dynamic CR. Whereas targeting static CR reduces the program size, we aim to improve energy efficiency by reducing the number of bits fetched, even with the expense of increasing code size. As loops are hot spots in program execution, we do not program the dictionaries during their execution in order not to cause excessive performance overheads. Instead, the dictionaries are programmed upon entering a loop region. These loop regions are decided during compilation based on a static CFG analysis.

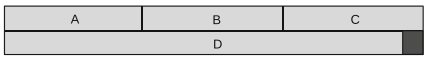
In the following, our code compression algorithm and dictionary design choices are discussed in detail.

4.1 Bundling

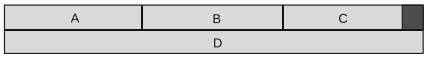
Figure 2 describes instruction alignment alternatives in schemes where the amount of different instruction lengths is limited to two. To avoid the complexity of variable-length decompression, we place compressed instructions as bundles into the memory, as illustrated in Fig. 2c. This allows interleaving compressed and uncompressed instructions, yet guaranteeing fixed alignment of the compressed instructions. In our approach, a bundle cannot contain fewer instructions than its defined size. The bundles are sized to fit into the uncompressed instruction length. This allows uncompressed instructions to be accessible in one fetch, as opposed to the alternative in Fig. 2a. Also, it allows the uncompressed instruction length to remain unchanged, as opposed to the alternative in Fig. 2b. This is done for convenience, as many ISAs utilize instructions whose lengths are a power of two, allowing straightforward interaction with memory components that are similarly sized. Because we require instructions to be



(a) Instruction alignment to compressed length. Uncompressed instruction D is divided into multiple fetch words i.e. D_0, D_1 and D_2 in this example.



(b) Instruction alignment to a multiple of compressed instructions.



(c) Instruction alignment to uncompressed length.

Fig. 2 Different instruction alignment options for compressed instructions. **A**, **B** and **C** are compressed instructions, whereas **D** is an uncompressed instruction. Padding is coloured black

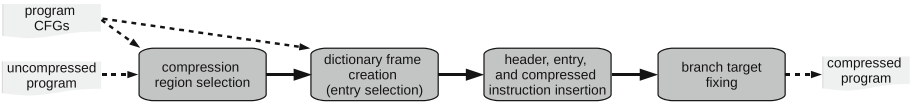


Fig. 3 Compression flow of of our method. Based on a static CFG analysis, compression regions and dictionary frames are created, after which special instructions are inserted to control the dictionary compression. As a last step, branch target addresses are fixed due to code size changes

compressed as bundles, the same instruction can occur in both uncompressed and compressed forms in the program, and even inside one BB.

4.2 Selective bundle-based code compression

Figure 3 presents the compression flow of our method. Although the method can be integrated into a compiler, it is described here as a post-pass, with the assumption that the relevant program information is retained after compilation. The compression of a program begins by constructing a CFG for each of its functions. Next, BBs belonging to loops are grouped into *compression regions* similar to superblocks [33]. The compression regions are created out of separate nested loops in a program. In programs with complex loop structures, this results in relatively large compression regions. Assuming that the amount of non-identical instructions increases as the loop size increases, having large compression regions does not minimize the dynamic CR or energy consumption, but finding the optimal compression region granularity can be considered as a separate research topic and is left for future work.

Next, a *dictionary frame*, which refers to the entries corresponding to a compression region, is created for each compression region if it is beneficial for dynamic CR. To avoid bias due to data-dependent execution, we do not use dynamic profiling, but instead rely on static CFG analysis. Our bundle-aware compression algorithm used to create the dictionary frames is presented in Fig. 4. As inputs, it takes the compression regions based on the program CFGs, as well as the parameter n for the required number of compressed instructions per bundle. In this work, the value of n is determined as a by-product of a design space search as described later. The algorithm starts by processing each compression region on line 1. The algorithm selects dictionary entries from the instructions by starting from inner loops. If all of the loop's instructions fit into the dictionaries (lines 19–22), all required entries are selected and the next loop level is examined. This is continued until the dictionaries are full, or no more instructions can be compressed.

```
   Input compression regions
   Output dictionary frames
   n = consecutive instructions required for a bundle
 1: for all region in regions do
 2:   while dictionaryFrame not full and BBsToHandle do
 3:     while true do
 4:       groupValid = True
 5:       bestGroup = None
 6:       i, newEntriesRequired, bestScore = 0
 7:       while i < region.instructions.length do
 8:         candidateBBGroup = region.instructions[i:i+n-1]
 9:         for instruction in candidateBBGroup do
10:           if instruction not isFirstIn(candidateBBGroup) then
11:             break
12:           end if
13:           for all field in instruction do
14:             if field.encoding not in dictionaryFrame[fieldIdx] then
15:               newEntriesRequired += 1
16:             end if
17:           end for
18:         end for
19:         if candidateBBGroup.fitsInto(dictionaryFrame) then
20:           for all unique entry in candidateBBGroup.entries do
21:             numOccurrences += entry.occurrencesIn(region)
22:           end for
23:         else
24:           groupValid = False
25:         end if
26:         if groupValid then
27:           score = numOccurrences/newEntriesRequired
28:           if score < bestScore then
29:             bestGroup = candidateBBGroup
30:             bestScore = score
31:           end if
32:         end if
33:         i += n
34:       end while
35:       if bestGroup != None then
36:         dictionaryFrame.addEntriesFrom(bestGroup)
37:       else
38:         break
39:       end if
40:     end while
41:   end while
42: end for
```

Fig. 4 Pseudocode to create dictionary frames. The algorithm tries to first fit inner loops into dictionaries. In case a loop level does not fit, a heuristic is used to select beneficial entries

Upon encountering a loop level whose instructions do not fit into the dictionaries, we calculate an *occurrence-cost* score for each candidate bundle on line 27. When bundling is used in conjunction with parallel dictionaries, simply selecting the most occurring entries for each dictionary does not result in the most instructions compressed, as all the fields of an instruction have to be selected for it to be compressed. In this case, greedy selection of entries can in the worst case result in no instructions being compressed, although valid entries have been chosen for the dictionaries. This occurs when each bundle has an instruction which contains a field that is not selected as an entry, prohibiting the compression of that bundle. In addition, exhaustively searching for the best entries proved to be prohibitively time-consuming in our early experiments. The occurrence-cost score is representative of the number of entries in the compression region that can be represented if a candidate bundle is compressed, as well as the amount of new entries required in the dictionaries. To add new entries to the dictionaries, there is a cost of

$$cost_e = \sum_{n=1}^{N} r_n \tag{1}$$

```
Input D, F, targetPrograms
Output dictionarySizes

1: for all d in D × F do
2:    if bundleSize == 1 then
3:       continue
4:    end if
5:    for all program in targetPrograms do
6:       compressedProgram = compress(program, d)
7:       results.append(analyzeTrace(compressedProgram))
8:    end for
9:    programSetResults.append(geoMean(results))
10: end for
11: dictionarySizes = programSetResults.minEnergy()
```

Fig. 5 Selection of the number of entries for the parallel dictionaries presented as pseudocode. D = range of entry counts to sweep, F = individual dictionaries, d = cartesian product of D and F. With the constraint of uncompressed word length as the maximum length of a compressed bundle, a sweep is performed over all dictionary sizing combinations. If the combination is valid, dynamic traces for a set of target programs are used to calculate an energy cost function

where N is the number of consecutive instructions in a bundle and r_n is the number of new entries that would be added to the dictionaries. The *occurrence-cost* score is then calculated with

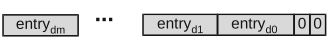
$$score_{oc} = \frac{cost_e}{s} \tag{2}$$

where $cost_e$ is calculated using Eq. 1 and s is the number of static occurrences in the whole region for the entry candidates in the bundle. On lines 28–30 and 35–39 in Fig. 4, the candidate bundle with the lowest occurrence-cost score is selected to be compressed, in case there is at least one valid bundle for compression.

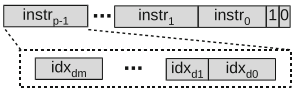
4.3 Parallel dictionary sizing

As the relationship between the number of dictionary entries, bundle size, dynamic compression ratio, and energy consumption is not trivial, we decide the parallel dictionary sizes by exhaustively sweeping the combinations of dictionary sizes. As a starting point, we assume that the instructions have been divided into fields, each corresponding to a dictionary. The sweeping process is described as pseudocode in Fig. 5. As input, the algorithm takes a set of target programs, a range of entry amounts to sweep, and the instruction bit indices that each dictionary compresses. On line 1, the cartesian product over the valid range of the number of entries to sweep D and the individual dictionaries F is performed. Combinations resulting in only one compressed instruction per bundle are discarded on lines 2 and 3. As we always align bundles to uncompressed instruction length, a single compressed instruction per bundle would not be beneficial. A set of target applications is iterated over in line 5. In line 6, the program is compressed, during which the algorithm described in Fig. 4 is used. As a parameter the sweeping algorithm takes d, the combination of entry counts for each dictionary. The energy consumption of the compressed program is analyzed using a dynamic trace. On line 9, the geometric mean of the program set is calculated. Although we chose geometric mean in order not to let outlier programs bias the compression, other types of means could be used depending on the desired behaviour. After sweeping across all the combinations, the entry amount combination resulting in the lowest energy consumption is selected on line 11.

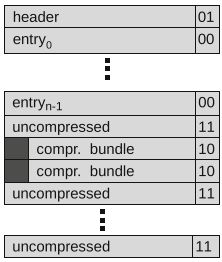
(a) Header instruction, which starts the programming of a new dictionary frame. It indicates the number of entries after it to be written into the dictionaries.



(b) An entry, which is used to program m dictionaries in parallel.



(c) A bundle of p compressed instructions. A compressed instruction contains an index to each of the parallel dictionaries.



(d) Example of instruction layout in memory. A header instruction indicates that the following n entries are to be programmed into the dictionaries. Compressed instructions are interleaved with uncompressed instructions. Two bits are used to indicate the type of instruction.

Fig. 6 **a** Special instructions used to control dictionary programming, **b**, **c** compressed instructions, and **d** an example of placement in memory

4.4 Programmable dictionary control

During execution, a special *header* instruction shown in Fig. 6a is used to start the programming of dictionaries. The instruction indicates the number of entries to sequentially program into the dictionaries, and writing always starts from the first index. Upon encountering a header, the current content of the dictionaries is considered invalid. In case there are fewer entries in a dictionary than indicated by the header, that dictionary is not written into after reaching the maximum number of entries. As the dictionary decompressor is naturally placed before the instruction decoder, the header instruction is handled inside the decompressor, and is not passed to the instruction decoder. The entries to be programmed shown in Fig. 6b are placed after the header, as illustrated in Fig. 6d. As the length of the entries is the same as the uncompressed instruction length, each of them requires a single fetch. In our approach, the processor is stalled while the dictionaries are programmed. After the entry count indicated by the header is reached, execution continues. The figure shows an example where the code contains uncompressed instructions that are executed as such, and compressed instructions, which are effectively indices to the dictionary entries.

As the header instruction encoding can be customized per target processor, it can be implemented by adding bits to the instruction words, or by using free ISA encodings. Figure 6 shows an example of the first case, where two bits of an instruction are used to identify the instruction type. The same bits are also used to identify compressed and uncompressed instructions.

4.5 Design considerations

By deciding the entries to compress based on a static profile, our method aims not to be biased due to data-dependent control flows in profiling runs. However, using dynamic profiling would likely be beneficial in cases where the control flow is more predictable and not data-dependent. For example, complex loop structures with multiple inner loops that have static iteration counts are currently not considered. In these cases, more frequently executed loops could be favoured when selecting entries for compression, instead of compressing the whole nested loop.

After compression, immediate values used for branch target addresses require fixing. For uncompressed instructions, this is trivial. However, in ISAs where the same bit indices can represent immediate values or instruction fields depending on the instruction template, compressing them poses a problem: (Part of) an immediate value used for branch addressing cannot share a dictionary entry with other instructions, because fixing it would change that entry. This would also change the bit pattern for the instruction sharing the entry. Thus, it is better to include instruction metadata in the compression, and treat branch immediate entries as always requiring new entries from dictionaries. Similarly, PC-relative branching requires consideration, as code compression can change the target addresses.

In our compression method, the entries to be programmed are always located after a header instruction. Another option would be to store the entries in a dedicated memory region, from where they would be fetched upon encountering a header instruction. This might be useful in a scheme, where the dictionaries would be programmed in a more fine-grained manner than the current approach. For example, if encountering the same dictionary frame that is currently stored, a branch over the entries would be required when they are stored directly after the header.

In our approach, execution is stalled while programming the dictionaries, as the entries are located in the program code after the header. While the amount of stall cycles could likely be reduced with prefetching schemes, these would not likely results in significant performance improvements, as the dictionaries are programmed relatively seldom. However, the effect of prefetching may be more significant, if more fine-grained compression region strategies are used.

Also, the number of instructions in a bundle is fixed at design time, and we only allow one bundle size in addition to uncompressed instructions. Although allowing variable-length bundles may improve the compression, studying the optimal amount of bundle sizes as well as selection of entries in this scheme is outside of the scope of this article.

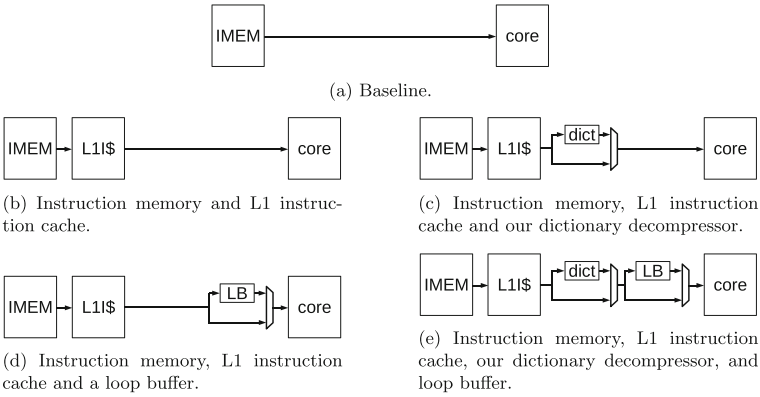
5 Instruction stream design by analytical modeling

To minimize the instruction stream energy consumption, its components must be selected and sized according to the expected target applications. Overly large components lead to poor utilization and energy overheads, whereas excessively small components cannot contain working sets of applications and require constantly changing their contents. Due to the large number of combinations resulting from different components and their sizing, implementing and verifying them on logic level becomes excessively laborous. To explore the instruction memory hierarchy design space, we construct an analytical energy consumption model of instruction stream components, similar to previous works [38, 39].

The system configurations used in the instruction stream design space exploration are illustrated in Fig. 7. The baseline system in Fig. 7a consists of a processor core and an SRAM main instruction memory (IMEM). Other components included in the model are an L1 instruction cache (L1I$), the decompressor of our compression method (dict), and a loop buffer (LB).

Energy consumption is obtained using the access counts per memory level multiplied by the energy per access of the respective memory level. The total system energy consumption is obtained with

$$E_{total} = E_{core} + E_{IMEM} + E_{L1I\$} + E_{dict} + E_{LB} \tag{3}$$



(a) Baseline.

(b) Instruction memory and L1 instruction cache.

(c) Instruction memory, L1 instruction cache and our dictionary decompressor.

(d) Instruction memory, L1 instruction cache and a loop buffer.

(e) Instruction memory, L1 instruction cache, our dictionary decompressor, and loop buffer.

Fig. 7 Components and configurations used in the analytical modeling of instruction stream hierarchies

where the energy consumption of components in Fig. 7 are represented. If a component is not used in the configuration, its energy consumption is set to zero. For the targeted embedded scenarios, we assume that the main instruction memory content is not changed during execution. The loop buffer and programmable dictionary are always written only before starting loop execution.

Computing energy for baseline configuration: For a given application, we compute the processor core energy E_{core} with total number of cycles and average energy per cycle as:

$$E_{core} = total\_cycles \cdot energy\_per\_cycle$$

The average energy per cycle does not take into account different program phases, but is the result of total energy consumed in an application divided by the application cycle count.

We obtain the IMEM energy consumption E_{IMEM} by retrieving the number of accesses (reads) and multiplying it by an energy per read value:

$$E_{IMEM} = IMEM\_accesses \cdot energy\_per\_read$$

Computing energy for the L1 cache: For a given application, we estimate the total cache energy with the number of cache accesses and miss count for a particular cache configuration. We assume that the energy consumed for a miss and a hit is known. The energy consumption is computed as:

$$\begin{aligned} E_{L1I\$} = {} & (L1\_accesses - L1\_misses) \cdot energy\_per\_hit \\ & + L1\_misses \cdot energy\_per\_miss \end{aligned}$$

Computing energy for the loop buffer: For calculating the LB energy consumption, we assume that the energy per active cycle (execution from LB), energy when filling the LB, and the idle energy are known. We calculate the total LB energy consumption with these three values and the cycle count corresponding to each:

$$\begin{aligned} E_{LB} = {} & LB\_active\_cycles \cdot energy\_per\_active\_cycle \\ & + LB\_fill\_cycles \cdot energy\_per\_fill\_cycle \\ & + (total\_cycles - LB\_active\_cycles - LB\_fill\_cycles) \\ & \cdot energy\_per\_idle\_cycle \end{aligned}$$

Computing energy for the programmable dictionary: To obtain the energy consumption of decompression, we divide program execution cycles into three subcomponents similar to the loop buffer: Active, fill, and idle. We assume that the energy consumption for each is known. The total energy consumption E_{dict} is then obtained by multiplying each value with the corresponding number of cycles and adding the results together.

$$\begin{aligned} E_{dict} = & idict\_active\_cycles \cdot energy\_per\_active\_cycle \\ & + idict\_fill\_cycles \cdot energy\_per\_fill\_cycle + (total\_cycles \\ & - idict\_active\_cycles - idict\_fill\_cycles) \cdot energy\_per\_idle\_cycle \end{aligned}$$

6 Evaluation

In our previous work [27] we evaluated our proposed code compression on a single-issue RISC-V processor, where it allowed a 5.5% reduction in system energy consumption on average and 21% in the best case. Although the previous state-of-the-art in programmable dictionary compression achieved better dynamic compression ratios, it had a large runtime overhead for programming dictionary entries, whereas the overhead was negligible with our proposed method. In this section we extend our work and evaluate the applicability of the method to statically scheduled multi-issue architectures. To this end, we implement our method in a *transport triggered architecture* (TTA) [40] processor. TTA is an exposed datapath architecture, where the programmer explicitly controls the transportation of operation input and output operands on data buses. An issue slot is reserved in the instruction word for each data bus. The TTA processor is selected as the evaluation platform as it can be considered as a superset of VLIW architectures. Both are statically scheduled multi-issue architectures with TTA having more degrees of freedom in operand movement due to the explicit datapath, and a VLIW can be modeled by constraining a TTA processor. The architectural details relevant to this work are listed in Table 2. For the purposes of this work, the TTA programming details are omitted, as they are not relevant.

Although previous work [17] has presented heuristics to group instruction bit indices to be compressed in parallel dictionaries, it was out of the scope of this article. For the evaluation of our approach, we manually selected the instruction bit groups for each parallel dictionary by observing the instruction template of the TTA processor. More precisely, the bits representing each issue slot were compressed separately. These are shown in Fig. 8. The *imm ctrl* field is used to select an instruction template between either a "regular" instruction or a 32-bit immediate value, and was grouped with issue slot 4. Although this coarse-grained grouping may not result in the optimal organization, it is an instinctive grouping and used here to illustrate the effectiveness of our approach as we show later.

The TTA instruction length is designed to be 64 bits wide after adding the two bits required for the dictionary control as described in Fig. 6. The instruction length is formed by first selecting the desired operations for the architecture and tuning the interconnection network between the function units to reach a desired level of performance. In our TTA model, the instruction length increases by adding operations and connections between function units. The issue width described in Table 2 is also a product of this exploration. Then, with additional operations, increasing immediate value widths, and further tuning the interconnection network, the instruction length is set to the nearest power of two to provide future readiness for integration with common memory components.

Table 2 TTA core characteristics

Issue slots	Instruction length	ISA	Scheduling	Control
5	64b	Custom	Static	Unconditional branches + Predicated execution

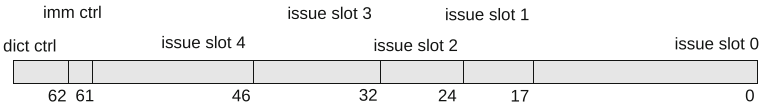


Fig. 8 TTA instruction template. Each "regular" instruction has five issue slots. By using the two immediate control bits, 32-bit immediates can be issued by using a combination of the issue slots. Two bits are used for controlling the dictionary compression. The TTA processor's issue slots present a natural grouping for instruction bit indices, when forming the parallel dictionaries.Would be nice to show how the bits are grouped into dictionaries

In addition to the CHStone [41] benchmarks used for evaluations in our previous work [27], we evaluate the compression on the TTA processor with Embench [42] benchmark suite. Seven out of 22 benchmarks were used, as the rest had qualities not suitable for our evaluations. Two of the benchmarks use recursive functions, which the current compression algorithm does not handle due to the lack of whole-program and call graph analysis. The other discarded benchmarks had 64-bit datatypes that are not supported in our current compiler, and floating point data types, whose soft emulation would not produce representative results.

Correct functionality of our approach is verified by compressing the evaluated benchmark programs, implementing the decompression hardware in RTL and simulating the design in ModelSim and comparing the traces produced by the executed programs to reference runs. The loop buffer (LB) used in evaluations is software-controlled, uses zero-overhead looping and supports single-level loops with no control flow inside. It is verified in the same fashion as the decompressor hardware. Designing the TTA core and generating RTL for it, as well as compiling target applications and producing the reference run traces from ISA simulations is done with OpenASIP, formerly known as *TTA-based Co-design Environment* (TCE) [43].

Synopsys Design Compiler is used to synthesize the designs on a 28 nm process. Power consumption is estimated using *switching activity information files* (SAIFs) produced by simulating the post-synthesis netlists in ModelSim. Access energy consumption for the SRAM instruction memory is obtained from CACTI [44]. Latch-based register file memories were used to store the entries for dictionaries and loop buffers.

6.1 Instruction memory hierarchy design

To select the system configuration for further energy consumption evaluations, we used the analytical model introduced in Sect. 5. Values obtained from the post-synthesis netlist simulations fed into the analytical model as access energy consumption estimates are illustrated in Fig. 9. Here, the L1 cache energy consumption was the largest of the components at all capacities. The dictionary and LB components had similar access energy consumption at small capacities, but the dictionary started consuming more energy when increasing the capacity. This was due to the addressing logic required for each parallel dictionary. Here the decompressor had five parallel dictionaries, one for each TTA issue slot.

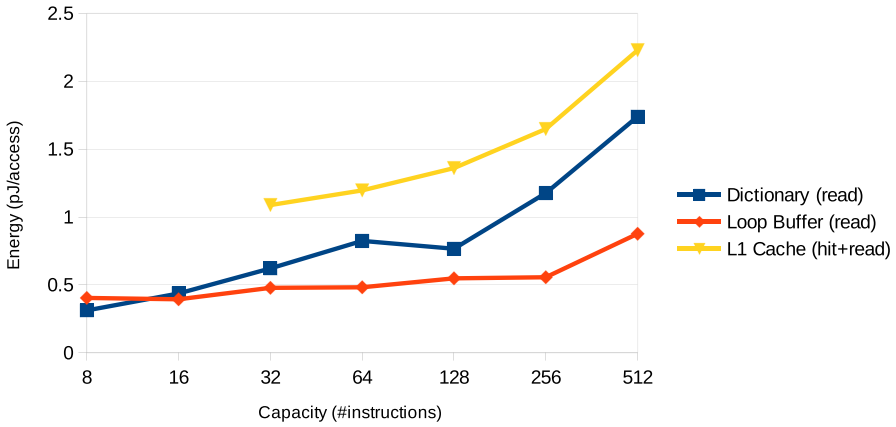


Fig. 9 Average energy per access of different instruction memory structures using post-synthesis netlist simulations in the 28 nm technology (0.9 V, 25 C, 500 MHz). These values were used in the analytical modeling. The entry size is 64-bit for all memory structures except the dictionary, where 5 sub-words are fetched from the dictionaries and combined to a single instruction

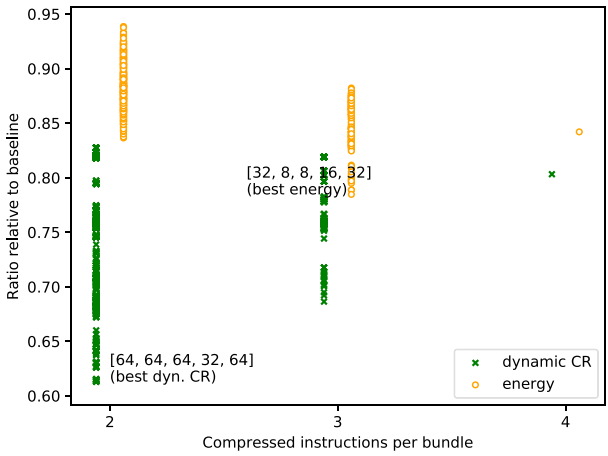
We assume the size of the main instruction memory *IMEM* to be 8192 lines (64-bit wide), as this was the minimum size to fit all of the evaluated benchmarks. The memory consumes approximately 14.45 pJ/access according to CACTI.

To select a reasonable instruction memory hierarchy, we first sized the L1 instruction cache, as this was considered a suitable approach by Rawlins and Gordon-Ross in schemes using code compression and loop buffering in conjunction [34]. We assumed a common cache line size of 16 B, and the cache was direct-mapped. Instruction traces for the CHStone and Embench benchmark sets were obtained using the OpenASIP toolset ISA simulator. For these applications, the cache with 256 lines (4 KiB) resulted in the lowest energy consumption when taking the average over the benchmark set. The energy consumption for this benchmark set is not reduced on average when using more than 256 cache lines due to the increased access cost and insufficient improvements in hit rates. In *aes, gsm, motion, and sha* a cache with 128 lines resulted in lower energy consumption.

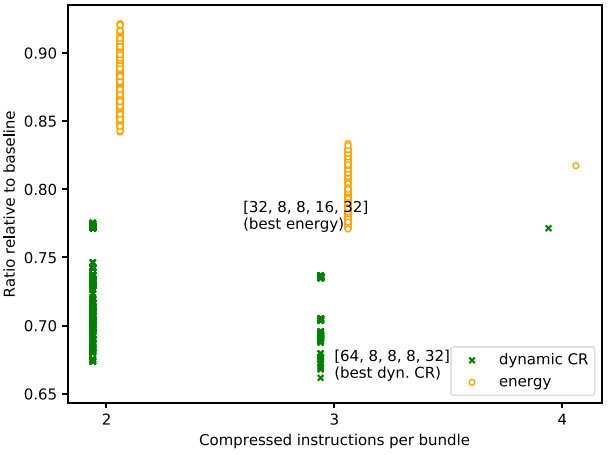
6.2 Dictionary sizing

After fixing the L1 instruction cache parameters, the parallel dictionary sizes (number of entries) were decided using the sweeping algorithm described in Fig. 5. The results are obtained by sweeping across dictionaries with 8, 16, 32 and 64 entries and using the analytical model to obtain access energy consumption. As a constraint, the compressed bundles were aligned to the uncompressed instruction length of the TTA processor. In this case, the instruction length is 62, to which two dictionary control bits are concatenated, forming a 64-bit final instruction. As explained in Fig. 6c, the number of compressed instructions in a bundle is obtained by adding the number of indexing bits required to index the parallel dictionaries, and checking how many compressed instructions can be bundled with the length constraint. If required, the compressed bundles are padded to meet the uncompressed instruction length, and bundles with size of one are not allowed, as this would worsen the compression ratios.

Figure 10a and b show the results of the dictionary sizing sweep. In the two figures, each benchmark set was used to calculate the dynamic CR and energy consumption values.



(a) CHStone, cache size 256 lines.

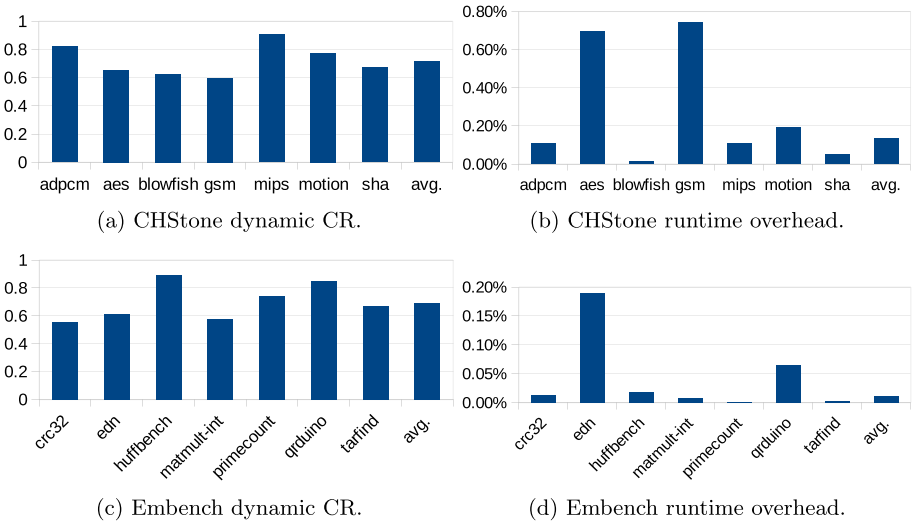


(b) Embench, cache size 256 lines.

Fig. 10 Results of bundle size and dictionary size sweep. Lower is better. The best individual entry amounts for dynamic CR and energy consumption annotated inside square brackets. Dynamic CR is the best at bundle size two, as this allows relatively large dictionary sizes to accommodate more of different entries, and compressible bundles are more likely with smaller bundle sizes. However, the lowest energy consumption is obtained with a larger bundle size, as this decreases the number of cache accesses and allows smaller dictionary size

The figures illustrate the relationship of the number of compressed instructions per bundle, number of dictionary entries, dynamic CR and energy consumption. Different data points illustrate the range of results obtained for the configurations. In the figures, a lower value is better, and the individual dictionary sizes for the best cases of dynamic CR and energy are annotated inside the square brackets.

In CHStone, minimal dynamic CR and energy consumption are obtained at different bundle sizes. Best dynamic CR, 0.61, is reached at bundle size two, and increases in relation to the bundle size. As our approach can only compress full bundles of consecutive instructions, the compression algorithm is more likely to find valid bundles at small bundle sizes. However, energy consumption is not minimized with the best dynamic CR. Remember that we set a



(a) CHStone dynamic CR. (b) CHStone runtime overhead.

(c) Embench dynamic CR. (d) Embench runtime overhead.

Fig. 11 Dynamic CR and runtime overhead on TTA processor. Results are relative to uncompressed execution. Lower is better

constraint for the bundle to fit into the uncompressed instruction length. This translates to more entries per dictionary at small bundle sizes, thus allowing more instructions to be compressed. The dictionary access energy depends on the number of entries, resulting in relatively high energy consumption at the small bundle sizes. The minimum energy consumption is reached at bundle size of three in Fig. 10a, which is a sweet spot in terms of dictionary entry amounts, instructions that can be compressed, and the dictionary access energy. Here the dynamic CR is 0.71. Although at bundle size 4 the dictionary size and access costs are relatively low, the number of valid bundles found by the compression algorithm is also low, resulting in poor dynamic CR.

Similar relationship between energy consumption and bundle size is seen in Embench, where bundle size 3 results in the best energy consumption. However, bundle size 3 also results in the best dynamic CR of 0.69. Although there are no large differences in the number of instructions compressed at bundle size 2 and 3, the larger bundle size requires fewer bits to be read from the next level in instruction stream hierarchy, resulting in lower dynamic CR.

Based on the results of the sweep, the 5 dictionaries for the TTA processor were selected to have 32, 8, 8, 16, and 32 entries, and a bundle size of three instructions was chosen as these resulted in the lowest energy consumption in the analytical model.

If there is time, analyze and illustrate the size of compression regions and the loops in them, so that we can motivate future work (better compression region analysis)

6.3 Compression ratios and performance

Results for dynamic CR in CHStone and Embench are presented in Fig. 11a and c. Our compression method results in a dynamic CR of 0.71 on average for CHStone and 0.69 for Embench, which are similar to the values obtained earlier for a single-issue RISC-V core [27]. *Mips* and *Huffbench* result in the worst dynamic CR, as both benchmarks consist of a large loop containing a large number of instructions. As a result, a relatively small

Table 3 Static CR results when using our compression method on the TTA processor with CHStone and Embench benchmarks

CHStone	Adpcm	Aes	Blowfish	GSM	Mips	Motion	SHA	Avg
	1.01	1.00	0.71	1.07	0.99	1.09	1.06	0.99
Embench	Tarfind	Qrduino	Primecount	Matmult-int	Huffbench	Edn	crc32	Avg
	1.01	0.98	1.05	0.99	0.97	1.09	1.10	1.03

portion of the frequently executed loop instructions are compressed. The best dynamic CR, 0.56, is obtained in *crc32*, where 69% of all executed instructions are compressed. This is explained by the straightforward structure of the benchmark, where a two-level nested loop constitutes the benchmark body. The inner loop has 13 instructions, of which 9 are compressed. Benchmarks resulting in high dynamic CR due to a large number of instructions in their frequently executed loops suggest that a more fine-grained program analysis and compression could lead to lower dynamic CRs.

To estimate the difference in compression ratios resulting from our method and optimal compression, we can assume that in the ideal case all instructions in a program would be compressed. To simplify, we can assume that the number of headers and entries is negligible, although in reality their number depends on the instructions mix, as well as number and size of dictionary frames. For the sake of this estimation, we can also assume that the number of BB instructions in compression regions always align perfectly with the bundle size. With these assumptions and the bundle size of three that was selected for the TTA processor, the ideal dynamic CR is 0.33. Although the value is a crude simplification, the large difference between it and the obtained results shows, that there is room to improve in regards to the compression algorithm. The difference between the ideal dynamic CR and the obtainable dynamic CR when accounting the headers and entries can be more realistically estimated when more fine-grained dictionary programming is studied, and is left as future work.

The dictionary programming runtime overhead on the total runtime in cycles is shown in Fig. 11b. The average runtime overhead is 0.13% in CHStone. This indicates that the dictionary programming overhead has a negligible impact on performance. *Blowfish* has a very low runtime overhead yet relatively low dynamic CR due to a large amount of instructions being represented by a relatively small amount of dictionary entries. Embench similarly results in negligible runtime overheads in all the benchmarks as seen in Fig. 11d.

Static CRs are presented in Table 3, where lower is better. Even though the main target of our approach is reducing the dynamic CR, the static CR (code size) is also reduced by 1% on average in CHStone. This is explained by the large compression regions and relatively small amount of dictionary entries. In addition, our approach currently targets nested loops, and a small part of the code can end up constituting most of the runtime. In Embench, the code size was increased by 3% on average. The code size is increased, when the number of inserted headers and dictionary entries outweighs the reductions caused by the compressed instructions. *Crc32*, which has the lowest (best) dynamic CR, also has the highest static CR. This is due to the benchmark being small, and the good dynamic CR results enabled by the compression of only 9 instructions. The code size is increased because there is little reuse of entries inside the dictionary frame. That is, the instructions have a diverse instruction mix.

Blowfish has a low static CR for the same reason as it has a low dynamic CR: relatively many instructions are compressed by relatively few entries. In addition, those instructions constitute a large part of the whole code.

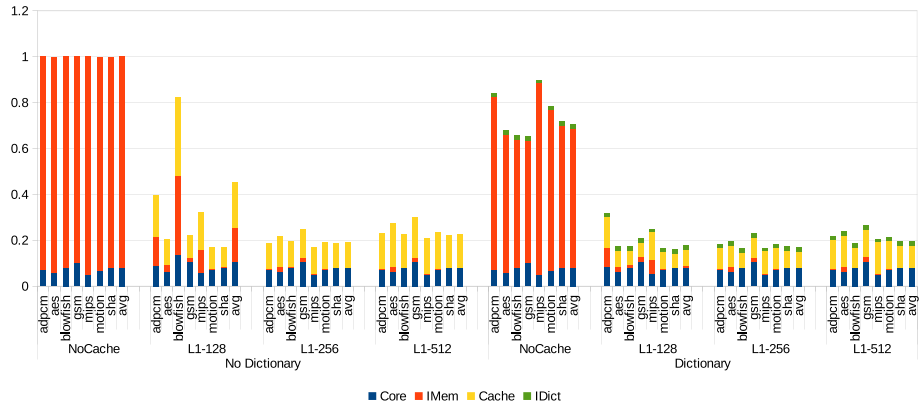


Fig. 12 TTA core energy consumption in CHStone benchmarks, relative to baseline system

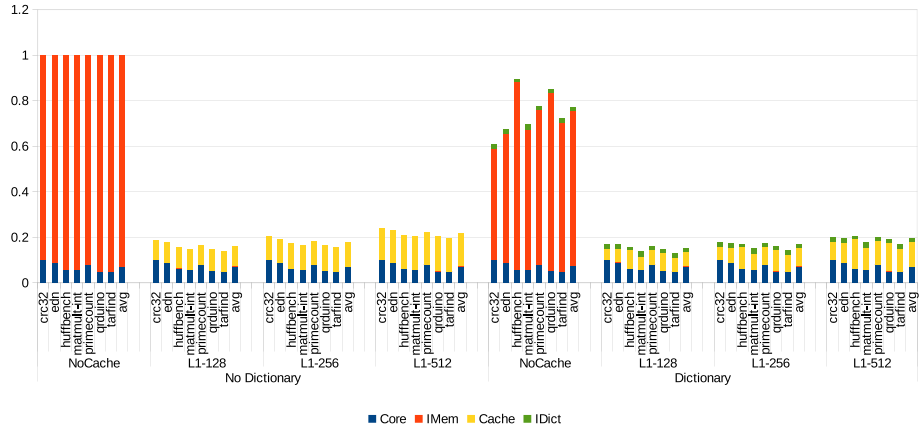


Fig. 13 TTA core energy consumption in Embench benchmarks, relative to baseline system

6.4 Energy consumption

Figure 12 and 13 present the energy consumption for the TTA core on the CHStone and Embench benchmarks. Remember that in Sect. 6.1 we used the analytical instruction stream model to conclude that a 256-line cache resulted in the lowest energy consumption on average when compared to a system with only an instruction memory and the processor. As exploring the full instruction stream design space exhaustively would have been excessively time-consuming, the design process in Sect. 6.1 started by deciding the cache size only. Thus, the 256-line cache was chosen and used when selecting the individual parallel dictionary sizes. However, at this point we can examine the behaviour also on other cache sizes. It follows from the L1 cache exploration that, on average, a 256-line cache leads to the lowest energy consumption for CHStone, while Embench benchmark performs best with a 128-line cache, due to the absence of loops with large bodies.

On average, adding an L1 cache to the system reduces the system energy consumption to 0.192 and 0.160 respectively, compared to the baseline. The addition of our proposed dictionary compression further reduces the energy consumption from 0.192 and 0.160 to

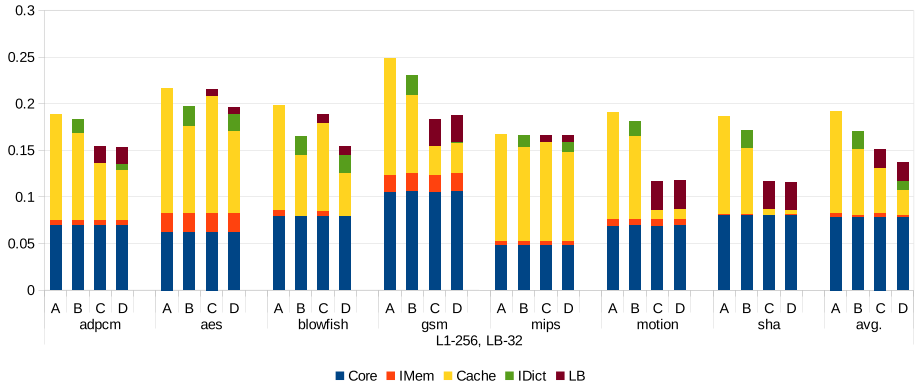


Fig. 14 System-level energy breakdown relative to baseline for CHStone benchmarks for different system configurations as depicted in Fig. 7 (configuration **A**: L1 cache, **B**: L1 cache + IDict, **C**: L1 cache + LB, **D**: L1 cache + IDict + LB)

0.170 and 0.154, respectively, an average reduction of 11.4% and 3.8% over just an L1 cache without dictionary decompression. The largest energy savings for the CHStone benchmarks in the L1-256 configuration are obtained for the *aes* (9.1%), *blowfish* (16.9%) and *sha* (8.2%) applications, due to their high dynamic CRs. Despite its high dynamic CR, the *gsm* energy savings are slightly lower (7.0%), which can be attributed to its higher L1 cache miss rate. The mips benchmark does not significantly benefit from dictionary compression due to its low dynamic CR resulting from its complex loop structure as discussed earlier.

The largest energy savings for the Embench benchmarks in the L1-128 configuration are obtained for the *crc32* (9.1%), *edn* (6.4%), and *tarfind* (6.3%) applications, due to their high dynamic compression ratios. *Huffbench* and *qrduino* hardly benefit from the dictionary decompression due to their low dynamic CR.

Despite L1-256 being the energy-optimal configuration for CHStone, on average, Fig. 12 shows some significant dictionary energy reductions for *adpcm*, *blowfish* and *mips* in the L1-128 configuration. When observing the cache performance, it turns out that L1-128 leads to a significant number of cache misses for *adpcm* (13.5%), *blowfish* (37.1%) and *mips* (10.5%), while cache misses in the other benchmarks are all in the range of 0.03%−3.1%. Using a dictionary reduces the number of cache misses significantly, due to fewer instructions being fetched, and for *blowfish* the miss rate is also significantly reduced from 37.1% to 2.13%, caused by the working set size reduction. This suggests, that in some cases our approach allows to increase the effective size of the cache by fitting more instructions into it.

6.5 Adding a loop buffer

In this section we discuss the effects of implementing our method in a processor that utilizes a loop buffer. This is a common instruction energy and performance optimization [45, 46]. Therefore, we will evaluate the energy efficiency of the TTA core with a loop buffer, as depicted in Fig. 7d and e. It is important to note that the loop buffer will decrease the effectiveness of the dictionary on straightforward loops. This is because our compression currently prioritizes inner loops, for which the LB is also used. The loop buffer also has a lower access energy as illustrated in Fig. 9, as it is more simple by design, compared to a

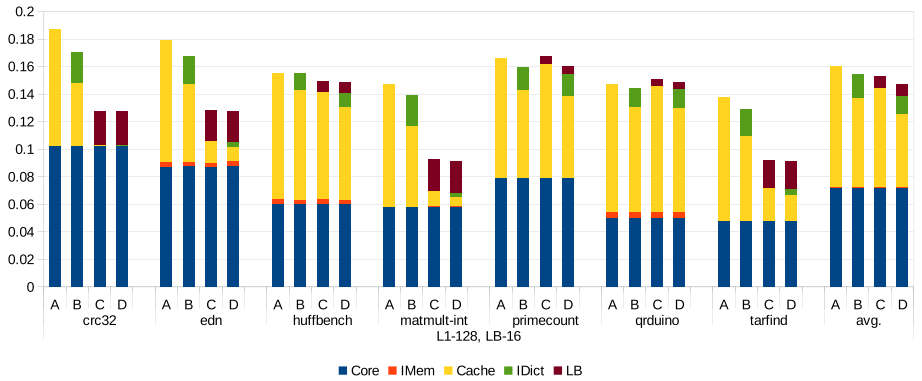


Fig. 15 System-level energy breakdown relative to baseline for Embench benchmarks for different system configurations as depicted in Fig. 7 (configuration **A**: L1 cache, **B**: L1 cache + IDict, **C**: L1 cache + LB, **D**: L1 cache + IDict + LB)

hardware-controlled cache and dictionary. Additionally, it saves the instruction fetch of the compressed instruction word that the dictionary has to perform.

Figures 14 and 15 depict the energy savings relative to the baseline for all system configurations while assuming instruction memory hierarchy parameters that had the lowest average energy consumption for each benchmark set. Configurations A and B correspond to the configurations with only an L1 cache and L1 cache with dictionary, respectively, as were depicted in Figs. 12 and 13. Configuration C considers an L1 cache with loop buffer, and configuration D combines the loop buffer with dictionary decompression.

For the CHStone benchmarks we consider the L1-256, LB-32 configuration. On average, extending configuration B with a loop buffer (B→D) leads to an additional 19.8% energy savings. However, the variation between benchmarks is quite significant. The LB is heavily utilised for the *gsm* (86.6%), *motion* (97.8%) and *sha* (94.6%) benchmarks, which leads to tremendous energy savings in the instruction delivery. It should be noted that for these benchmarks the loop buffer overlaps with the compressed code regions, which causes the minimal energy contribution of the dictionary in configuration D, compared to C. When the dictionary can be used for a significant part of the application, and the loop buffer usage is limited, as is the case for *aes* and *blowfish*, having a dictionary improves energy efficiency over just having a loop buffer. For *mips* the energy savings of both the loop buffer and dictionary are limited, as neither can be used effectively.

For the Embench benchmarks we consider the L1-128, LB-16 configuration, as most applications have smaller inner loops and working sets when compared to CHStone applications. On average, extending configuration B with a loop buffer (B→D) leads to an additional 4.5% energy savings. The LB is heavily utilized for *crc32* (99.7%), *edn* (84.5%), *matmult-int* (88.3%) and *tarfind* (69.5%). In these benchmarks the compressed loops mostly overlap with the LB. For *huffbench*, *primecount* and *qrduino* adding a dictionary improves the energy efficiency only slightly, due to the low dynamic CR. For *primecount* and *qrduino* a loop buffer cannot be effectively used, which results in a small energy penalty due to the idle overhead of the LB.

It should be noted that the current compression algorithm does not consider the LB as part of the system. This leads to compressing parts of the code that will end up in the LB anyway, thereby not optimally using the limited dictionary space.

As a conclusion, adding our dictionary compression scheme in conjunction with an already energy-saving LB can help in reducing instruction stream energy consumption in processors especially when the LB cannot be used efficiently for all loops. Although the LB is an energy-efficient component due to its simple nature and removal of loop iteration overheads, its use is limited in complex applications. From the perspective of our compression method, the presence of the LB should be taken into account when performing program compression, and the development of such compression algorithms requires further research.

6.6 Comparison with previous work

Table 4 presents comparison with previous work on code compression. The majority of previous research measures its effectiveness with static CR and impact on performance, whereas only a few works report dynamic CR. Similarly, the effect on energy consumption is typically not measured. Although the numbers reported in previous work have been produced in various system setups, the comparison provides ballpark numbers to the effectiveness of compression. The dynamic CR results are also similar on average. Although the dynamic CR reported by Thuresson et al. [25] is significantly lower than other methods, we previously observed [27] that large loops result in significant dictionary programming overheads in their method. As the dictionary entry amounts are fixed at design time and their method compresses all program instructions, dictionary programming is required at each loop iteration if the loop instructions require more entries than the dictionary has.

It is interesting to note that Seong et al. [15] and Shrivastava & Mishra [22] reported similar significant improvements in static CR, when a bitmask scheme was used in conjunction with dictionary compression. This suggests that there could be further gains to be achieved also for our approach. However, the authors did not measure the effect on dynamic CR which is the primary target of our method.

Previous works typically report execution time improvements due to improved cache hit ratios. However, this suggests that the evaluated caches are relatively small and result in relatively many misses during execution, or that the baseline compression method itself incurs a performance overhead which is then improved in the work. Unlike this work, they do not evaluate energy consumption. In our evaluation in an embedded system scenario we found that it is typically beneficial for energy consumption if the cache is large enough to accommodate frequently executed loops, as filling the cache from the next level in instruction stream hierarchy is expensive. Our compression method does not add significantly to program execution time when compared to the baseline system. In this light, the comparison of performance improvements is not meaningful as we reach the lowest energy consumption with already relatively large caches whose hit rates are not further improved with compression.

Although only a few works have evaluated the effect of compression on energy consumption, the reported results have been significant. Similar to our work, Benini et al. [19] observed that the improvements of code compression on dynamic CR are reduced in a systems containing an instruction cache, as the cache itself reduces memory traffic.

7 Conclusion

This paper presented our programmable code compression method, which reduces energy consumption by improving the dynamic compression ratio and reducing the amount of accesses to higher hierarchy levels of the instruction stream. Based on a literature review of

Table 4 Comparison of previous code compression methods. Reported results

	Static CR	Dynamic CR	Runtime w.r.t. baseline	Energy w.r.t baseline
CCRP [12]	0.73	0.67−0.86	0.75−1.10	
Lefurgy et al. [13]	0.61−0.74		1.0	
Benini et al. [23]	0.41−0.52			
Lekatsas et al. [18]			0.38−0.89	
Benini et al. [19]	0.59−1.28	0.25−1.3	0.38−1.98	0.35−1.9
Netto et al. [20]	0.71−0.88		0.78	
Das et al. [21]	0.70−0.90			
Brorsson & Collin [24]	0.67−0.79	0.50−0.80	1.08−1.12	0.50−0.91
Xie et al. [14]	0.56−0.83			
Lin et al. [17]	0.74−0.83			
Seong et al. [15]	0.55−0.65			
Thuresson et al. [25]	0.88−0.96	0.42	1.01	
Bonny & Henkel [16]	0.42−0.50		1.13−1.18	
Shrivastava & Mishra [22]	0.60−0.65		0.05−1.0	
This work	0.71−1.10	0.56−0.91	1.00−1.01	0.09−0.20

energy-efficient design approaches for code compression, we divided instructions into fields which are compressed individually using parallel dictionaries. We then grouped selectively compressed instructions into fixed-length bundles, and aligned them with uncompressed instructions. The contents of the parallel dictionaries were programmed during runtime by using special instructions. This was done outside of loop regions to avoid excessive performance overheads.

We also described an approach to select parallel dictionary sizes based on an analytical energy consumption model, and a method using the same analytical model to estimate the best instruction stream hierarchy in terms of energy consumption including an L1 instruction cache, a loop buffer, and our programmable dictionary compression.

To demonstrate applicability on different processor architectures, we evaluated our method on a VLIW processor in addition to a previous evaluation on a RISC-V single-issue processor. The VLIW evaluation was done using benchmarks from CHStone and Embench suites. Our approach resulted in dynamic compression ratios of 0.71 and 0.69 averaged for the benchmark suites, similar to the previously obtained RISC-V results. Unlike the state-of-the-art programmable dictionary compression, our method resulted in negligible runtime overheads while producing good dynamic compression ratios.

To evaluate the effect of instruction stream components, we chose a system consisting of an SRAM instruction memory and the VLIW core as the baseline. This configuration was extended with an L1 cache. On average, extending this system with a dictionary leads to 11.4% and 3.8% system energy savings, for the CHStone and Embench benchmarks, respectively. Adding a loop buffer improves the energy efficiency by an additional 19.8% and 4.5% on average for the CHStone and Embench, respectively. This complete instruction delivery optimization consisting of an L1 cache, dictionary compression, and loop buffer reduces the relative energy to 0.14 and 0.15 on average for the CHStone and Embench benchmarks, respectively. For simple applications consisting of straightforward loops, a loop buffer allows the instruction stream's proportion of total energy consumption to reach relatively low values. However, having dictionary compression and a loop buffer in a system helps in reducing

energy consumption in more complex applications containing loops where the loop buffer cannot be used.

Investigating the effects of whole-program analysis and more fine-grained compression region selection inside nested loops is left as future work. In addition, further research on compression algorithms is necessary in case of systems containing dictionary compression and a loop buffer. Our approach will also likely benefit from at least two complementing methods: Instruction re-encoding [16] and using bitmasks to indicate differences between similar instructions [15].

Funding Open access funding provided by Tampere University (including Tampere University Hospital). This work was supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware) and Academy of Finland (decision #331344).

Declarations

Conflict of interest The authors have no Conflict of interest to declare that are relevant to the content of this article.

References

1. Silven O, Jyrkkä K (2007) Observations on power-efficiency trends in mobile communication devices. EURASIP J Embedded Syst 2007(1)
2. Collin M, Brorsson M (2003) Low power instruction fetch using profiled variable length instructions. In: Proceedings of the international systems-on-chip conference (SoC), pp 183–188
3. Lee Y, Avizienis R, Bishara A, Xia R, Lockhart D, Batten C, Asanović K (2011) Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In: Proceedings of the international symposium on computer architecture (ISCA), pp 129–140
4. Schiavone D, Conti F, Rossi D, Gautschi M, Pullini A, Flamand E, Benini L (2017) Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for internet-of-things applications. In: Proceedings of international symposium on power and timing modeling, optimization and simulation (PATMOS)
5. Molendijk M, Putter F, Gomony M, Jääskeläinen P, Corporaal H (2023) BrainTTA: A 28.6 TOPS/W compiler programmable transport-triggered NN SoC. In: Proceedings of the international conference on computer design (ICCD), pp 78–85
6. Roman Kaplan, Michael Stearns: Intel® gaudi® 3 AI accelerator. White paper, Intel Corporation (April 2024). https://www.intel.com/content/www/us/en/content-details/817486/intel-gaudi-3-ai-accelerator-white-paper.html
7. AMD Inc.: AI engines and their applications. White paper (September 2022). https://www.xilinx.com/content/dam/xilinx/support/documents/white_papers/wp506-ai-engine.pdf
8. Kin J, Munish Gupta Mangione-Smith WH (1997) The filter cache: an energy efficient memory structure. In: Proceedings of the international symposium on microarchitecture (MICRO), pp 184–193
9. Hiraki M, Bajwa RS, Kojima H, Gorny DJ, Nitta K, Shri A (1996) Stage-skip pipeline: a low power processor architecture using a decoded instruction buffer. In: Proceedings of 1996 international symposium on low power electronics and design, pp 353–358
10. Lee L, Moyer B, Arends J (1999) Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In: Proceedings of the international symposium on low power electronics and design (ISLPED), pp 267–269

11. Uh G-R, Wang Y, Whalley D, Jinturkar S, Burns C, Cao V (1999) Effective exploitation of a zero overhead loop buffer. SIGPLAN Notices 34(7):10–19
12. Wolfe A, Chanin A (1992) Executing compressed programs on an embedded RISC architecture. In: Proceedings of the international symposium on microarchitecture (MICRO), pp. 81–91
13. Lefurgy C, Bird P, Chen I-C Mudge (1997) Improving code density using compression techniques. In: Proceedings of the international symposium on microarchitecture (MICRO), pp 194–203
14. Xie Y, Wolf W, Lekatsas H (2006) Code compression for embedded VLIW processors using variable-to-fixed coding. IEEE Trans Very Large Scale Integr Syst 14(5):525–536
15. Seong S, Mishra P (2008) Bitmask-based code compression for embedded systems. IEEE Trans Comput Aided Des Integr Circuits Syst 27(4):673–685
16. Bonny T, Henkel J (2010) Huffman-based code compression techniques for embedded processors. ACM Trans Design Autom Electron Syst 15(4):31–37
17. Lin CH, Xie Y, Wolf W (2007) Code compression for VLIW embedded systems using a self-generating table. Trans Very Large Scale Integr Syst 15(10)
18. Lekatsas H, Henkel J, Jakkula V (2002) Design of an one-cycle decompression hardware for performance increase in embedded systems. In: Proceedings of the design automation conference (DAC), pp. 34–39
19. Benini L, Menichelli F, Olivieri M (2004) A class of code compression schemes for reducing power consumption in embedded microprocessor systems. IEEE Trans Comput 53(4):467–482
20. Netto EW, Azevedo R, Centoducatte P, Araujo G (2004) Multi-profile instruction based compression. In: Proceedings of the symposium on computer architecture and high performance computing (SBAC-PAD), pp. 23–29
21. Das D, Kumar R, Chakrabarti PP (2005) Dictionary based code compression for variable length instruction encodings. In: Proceedings of the international conference on VLSI design held jointly with international conference on embedded systems design, pp 545–550
22. Shrivastava K, Mishra P (2011) Dual code compression for embedded systems. In: Proceedings of the international conference on VLSI design (VLSID), pp. 177–182
23. Benini L, Macii A, Macii E, Poncino M (1999) Selective instruction compression for memory energy reduction in embedded systems. In: Proceedings of the international symposium on low power electronics and design (ISLPED), pp 206–211
24. Brorsson M, Collin M (2006) Adaptive and flexible dictionary code compression for embedded applications. In: Proceedings of the international conference on compilers, architecture and synthesis for embedded systems
25. Thuresson M, Själander M, Stenstrom P (2009) A flexible code compression scheme using partitioned look-up tables. In: Proceedings of the international conference on high performance embedded architectures and compilers (HiPEAC), pp 95–109
26. Yoshida Y, Song B-Y, Okuhata H, Onoye T, Shirakawa I (1997) An object code compression approach to embedded processors. In: Proceedings of the international symposium on low power electronics and design (ISLPED), pp 265–268
27. Multanen J, Hepola K, Jääskeläinen P (2020) Programmable dictionary code compression for instruction stream energy efficiency. In: Proceedings of the international conference on computer design (ICCD), pp 356–363
28. Conte T, Banerjia S, Larin S, Menezes K, Sathaye S (1996) Instruction fetch mechanisms for VLIW architectures with compressed encodings. In: Proceedings of the international symposium on microarchitecture. (MICRO), pp 201–211
29. Lefurgy C, Piccininni E, Mudge T (1999) Evaluation of a high performance code compression method. In: Proceedings of the international symposium on microarchitecture (MICRO), pp 93–102
30. Ishiura N, Yamaguchi M (1997) Instruction code compression for application specific VLIW processors based on automatic field partitioning. In: Proceedings of international workshop on synthesis and system integration of mixed technologies (SASIMI)
31. Lekatsas H, Wolf W (1998) Code compression for embedded systems. In: Proceedings of the design automation conference (DAC), pp 516–521
32. Li L, Chakrabarty K, Touba N (2003) Test data compression using dictionaries with selective entries and fixed-length indices. Trans Design Autom Electron Syst 8(4):470–490
33. Hwu W-M, Mahlke S, Chen W, Chang P, Warter N, Bringmann R, Ouellette R, Hank R, Kiyohara T, Haab G, Holm J, Lavery D (1993) The superblock: an effective technique for VLIW and superscalar compilation. J Supercomput 7
34. Rawlins M, Gordon-Ross A (2011) On the interplay of loop caching, code compression, and cache configuration. In: Proceedings of the Asia and South pacific design automation conference (ASP-DAC), pp 243–248

35. Benes R, Nowick S, Wolfe A (1998) A fast asynchronous huffman decoder for compressed-code embedded processors. In: Proceedings of the fourth international symposium on advanced research in asynchronous circuits and systems (ASYNC), pp 43–56
36. Pan H, Asanović K (2001) Heads and tails: a variable-length instruction format supporting parallel fetch and decode. In: Proceedings of the international conference on compilers, architecture, and synthesis for embedded systems (CASES), pp 168–175
37. Lefurgy C, Piccininni E, Mudge T (2000) Reducing code size with run-time decompression. In: Proceedings of the international symposium on high-performance computer architecture (HPCA), pp 218–228
38. Gu J, Guo H, Ishihara T (2013) Dlic: Decoded loop instructions caching for energy-aware embedded processors. ACM Trans Embedded Comput Syst 13:1–23
39. Rawlins M, Gordon-Ross A (2013) Adaptive loop caching using lightweight runtime control flow analysis. ACM Trans Embedded Comput Syst 12:1–26
40. Corporaal H (1998) Microprocessor architectures: from VLIW to TTA. Wiley, Chichester, England
41. Hara Y, Tomiyama H, Honda S, Takada H (2009) Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. J Inf Process 17:242–254
42. Bennett J, Dabbelt P, Garlati C, Madhusudan G, Mudge T, Patterson D (2019) Embench: An evolving benchmark suite for embedded IoT computers from an academic-industrial cooperative
43. Jääskeläinen P, Viitanen T, Takala J, Berg H (2017) HW/SW co-design toolset for customization of exposed datapath processors. Comput Platforms Softw Defined Radio
44. Li S, Chen K, Ahn JH, Brockman J, Jouppi N (2011) CACTI-P: architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In: Proceedings of the international conference on computer-aided design, pp 694–701
45. Codrescu L, Anderson W, Venkumanhanti S, Zeng M, Plondke E, Koob C, Ingle A, Tabony C, Maule R (2014) Hexagon DSP: an architecture optimized for mobile multimedia and communications. IEEE Micro 34(2):34–43
46. Chen C, Xiang X, Liu C, Shang Y, Guo R, Liu D, Lu Y, Hao Z, Luo J, Chen Z, Li C, Pu Y, Meng J, Yan X, Xie Y, Qi X (2020) Xuantie-910: a commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension : industrial product. In: Proceedings of the international symposium on computer architecture (ISCA), pp 52–64