Programmable Dictionary Code Compression for Instruction Stream Energy Efficiency

Joonas Multanen, Kari Hepola, Pekka Jääskeläinen
Faculty of Information Technology and Communication Sciences (ITC)
Tampere University
Tampere, Finland
{joonas.multanen, kari.hepola, pekka.jaaskelainen}@tuni.fi

***Abstract*—We propose a novel instruction compression scheme based on fine-grained programmable dictionaries. In its core is a compile-time region-based control flow analysis to selectively update the dictionary contents at runtime, minimizing the update overheads, while maximizing the beneficial use of the dictionary slots. Unlike in the previous work, our approach selects regions of instructions to compress at compile time and changes dictionary contents in a fine-grained manner at runtime with the primary goal of reducing the energy footprint of the processor instruction stream.**

The proposed instruction compression scheme is evaluated using RISC-V as an example instruction set architecture. The energy savings are compared to an instruction scratch pad and a filter cache as the next level storage. The method reduces instruction stream energy consumption up to 21% and 5.5% on average when compared to the RISC-V C extension with a 1% runtime overhead and a negligible hardware overhead. The previous state-of-the-art programmable dictionary compression method provides a slightly better compression ratio, but induces about 30% runtime overhead.

I. Introduction

Although required for software-based programmability, the instruction stream of a processor is effectively an overhead in the big picture; it does not directly contribute to the processing of data in computation. Since the instruction stream in programmable processors can constitute up to 40% of total system energy [1], [2] and up to 70% of processor core power [3], it is appealing to focus on the minimization of its impact.

Previous work attempts to address the instruction stream overheads at various design hierarchy levels. In code compression, the aim is to reduce the bits required by the program. When minimizing *static compression ratio* (CR), redundancy in the program image can be utilized to save storage space in the instruction memory. On the other hand, if optimizing *dynamic CR*, the goal is to reduce the total number of *fetched* instruction bits in order to improve the cache hit ratio or to reduce bit toggling, eventually improving the energy efficiency of a software programmable processor. In that case, the focus of the compression is on the most executed instructions.

Previous code compression approaches mainly focus on static code size reduction with only a few works [4], [5] concentrating on dynamic CR. In this paper, we propose a code compression method targeted for energy efficiency by optimizing specifically for dynamic CR. Evaluated in an embedded system scenario, our approach achieves an average instruction stream energy reduction of 5.5% and 21% in the best case compared to RISC-V C extension [6] with an average runtime overhead of 1%. Although a state-of-the-art dictionary compression method [7] produces somewhat better dynamic compression ratios, its runtime overhead is significantly higher, degrading execution performance.

We identify the following key contributions in this paper:

- A novel *run-time programmable* code compression scheme,
- a static CFG analysis algorithm to group program basic blocks into *code compression regions*,
- a heuristic to find most suitable instruction bundles to place into dictionaries,
- a low-overhead decompression hardware architecture designed for minimal programming overhead, and
- the first evaluation of a dictionary compression scheme on the RISC-V ISA.

The rest of this paper is organized as follows. Section II overviews the relevant previous work on instruction compression. Section III introduces background on code compression and Section IV continues by describing the proposed method. In Section V, the proposed method is evaluated and results are presented. Finally, Section VI concludes the paper.

II. Previous Work

The proposed method requires design choices on the type of compression used, the strategies to choose the dictionary entries, and the granularity of the dynamic control over compression. Therefore, we review the previous work in these topics in the following subsections. The most relevant references are summarized in Table I.

A. Compression Types

The choice of compression type has significant impact on compression ratios, runtime overhead, area occupation and energy consumption. Although statistical methods such as Huffman coding and *run length encoding* (RLE) typically lead to better CR compared to dictionary-based methods, they result in a more complex decompression logic. Wolfe and Chanin [8] use compression based on Huffman code. During execution, encoded instructions are fetched into a cache and decoded

	type	update granularity	compression target	note	year
CCRP [8]	Huffman	program	static CR		1992
Lefurgy et al. [9]	dictionary	program	static CR	compresses instr. sequences	1997
Benini et al. [4]	dictionary	program	dyn. CR	dynamic profiling	1999
Lin et al. [10]	LZW	dynamic	static CR	on-the-fly generated code table	2007
bitmask [11]	dictionary + bitmask	program	static CR		2006
Brorsson & Collin [5]	dictionary	function call	dyn. CR + energy		2006
Thuresson et al. [7]	dictionary	basic block	dyn. CR		2009

TABLE I: Comparison of previous code compression methods.

for execution. Lin et al. [10] use *Lempel-Ziv-Welch* (LZW) to compress program *basic blocks* (BBs). During compression and decompression, a coding table is dynamically generated and cleared upon encountering a branch target.

ARM Thumb [12], MIPS16 [13] and RISC-V C extension (RVC) [6] are *reduced instruction sets*, where a subset of instructions are compressed by representing them with 16 bits instead of the regular 32 bits. For each of the two instruction modes, a separate decoding hardware is used.

Benini et al. [4] mix uncompressed (32-bit) and compressed (8-bit) instructions in the memory. A *mark* of 8 bits is used to indicate an uncompressed instruction in their dictionary scheme. Their results indicate that fetching uncompressed instructions in a single fetch is beneficial, which we adopt to our work. Lefurgy et al. [9] use dictionary compression to reduce code size. They form a single dictionary per program and align variable-length codewords to 4-bit boundaries. Lefurgy et al. [14] split 32-bit instructions into two parallel 32-entry dictionaries to compress instructions in the IBM PowerPC. In our work, we also split instructions to multiple dictionaries based on the instruction subfields of the target architecture ISA. Seong and Mishra [11] describe a *bitmask* scheme on top of dictionary compression. This allows multiple instructions to share a single entry, if the differences can be expressed with an XOR bitmask. This approach is orthogonal with dictionary compression and can be applied on top of the proposed method.

In the majority of the previous work, the preferred approach in recent instruction memory hierarchies incorporating instruction compression is dictionary compression [4], [7], [9], [11], [14], which we also base our work on.

B. Content Strategies

The method for finding optimal dictionary entries has been studied as an isolated research problem. Li and Chakrabarty [15] show that finding optimal instructions for dictionary compression is an *NP*-hard problem. Thus, previous work typically uses heuristics to select dictionary entries.

Benini et al. [4] observe that per program, the majority of the most executed instructions in their benchmark set fits into a dictionary of 256 instructions.

Lin et al. [10] analyze program *control flow graphs* (CFGs) to identify *branch blocks*, also referred to as *superblocks* [16], which are regions of *basic blocks* (BB) where a branch is only allowed to the first BB. Our proposed approach uses code regions similar to superblocks.

Ishiura and Yamaguchi [17] describe a field partitioning scheme for a *very long instruction word* (VLIW) architecture. They present a heuristic to approximate the best field partitions to be compressed individually. These are formed in parallel and iteratively merged in a greedy fashion.

To lower the dynamic cost of programming dictionaries, Thuresson et al. [7] search for identical entries between BBs and recursively push dictionary programming instructions to predecessor BBs, if the programming cost there is lower. The cost is determined by the amount of BB executions from a trace profile. The authors do not pack immediate values, which require fixing after compression. To avoid execution path bias, our proposed method uses static analysis instead of dynamic. Also, our algorithm handles immediate values and their correct execution is verified using RTL simulations.

C. Programming Granularity

While large dictionaries can accommodate more entries, accessing them is less energy efficient when compared to smaller ones. This brings a trade-off: How can dictionaries be kept small and energy-efficient, while maintaining good static and/or dynamic CR? A solution to this is dynamic programmability of dictionaries. However, this introduces additional challenges to the software side: At which point in the program to change the contents? In addition, the dictionary contents present an additional *architectural state*, which must be preserved across function calls, exceptions and context switches.

Previous work considers updating dictionaries at different levels of granularity during execution. Majority of the previous work updates dictionary contents at the start of a program [4], [8], [9], [11]. Brorsson & Collin [5] evaluate updating dictionaries at different granularities, of which the most fine-grained is at function-level. Their results suggest, that dynamic CR and energy consumption can be improved by updating dictionaries at context switches. However, as their approach compresses all instructions, loops that do not fit into dictionaries cause significant overheads in execution time and energy consumption. Clearly, compressing all instructions is harmful, when a benchmark exceeds the capacity of dictionaries whose size is fixed at design time.

Thuresson et al. [7] update dictionaries at the granularity of BBs. Upon entering a BB, dictionary entries can be programmed one-by-one. In their approach, all instructions are compressed, whereas in our proposed method only instructions considered beneficial for improving the energy footprint are compressed. Although the authors report execution time overheads of 1% from the dictionary programming, our evaluations show that for loops that do not fit into the dictionary, the

execution time overhead is significant. To address this, our method updates the dictionaries before loop regions.

Although relatively old, we consider the approach proposed by Thuresson et al. to still be the state-of-the-art in programmable dictionary compression. It achieves high compression ratios with a minor runtime overhead. Therefore, we use it as the baseline in our evaluation. It is also closest to the proposed method in terms of dictionary control granularity. Although there are more recent publications on instruction compression, such as [18], to the best of our knowledge they do not contribute on fine-grained programmable dictionaries.

III. Code Compression for Energy-Efficiency

In this section, we discuss the effects of dictionary compression design choices on energy-efficiency. Previous work on code compression has mostly concentrated on optimizing for static CR, while the potential for reducing energy consumption has received less attention. Our proposed method primarily targets dynamic compression ratio. Compared to previous work, our approach allows fine-grained dynamic control of dictionary contents, while keeping the runtime overhead low by not compressing all instructions. Ideally, in terms of energy-efficiency, the dictionaries should be as small as possible, while still providing a good dynamic CR. Compressing instructions requires fetching from both the main memory hierarchy as well as the dictionary, resulting in a higher total number of bits fetched compared to uncompressed instructions. However, as fewer bits are fetched from the expensive main memory, improvements in both energy consumption and performance are possible, since the dictionary accesses are assumed to be faster and consume less energy.

Here, we consider the dictionary or dictionaries to be programmable. Even though programmability can add complexity to the compression software, it can improve the compression ratio if instruction mixes vary significantly between different program phases [7].

A. Dictionary Sizing and Instruction Subfields

In order to decrease redundancy in instruction compression, instruction subfields such as immediate, opcode or register index fields can be compressed individually. Here, it makes sense to group together bits belonging to the same subfield [17]. As some subfields are not used in all instructions and some only use a small number of their possible bit combinations, these subfields can have good compression ratios even with relatively small dictionaries compared to the number of total bit combinations they can represent. For energy-efficiency, dictionaries should be kept as small as possible while still maintaining good compression ratios.

B. Should All Instructions Be Compressed?

In addition to programmability, the issue of limited amount of dictionary entries can be alleviated by allowing both uncompressed and compressed instructions in program code. Consider a region of code that is executed only once. If the entries between instructions in the region are not shared, the region should not be compressed; fetching and writing the entries into a dictionary in addition to fetching the compressed instructions for them only increases execution time and energy consumption. Another harmful case can be found in program inner loops: if all of the entries of a loop cannot fit into a dictionary and uncompressed instructions are not allowed, the dictionary will have to be programmed at each iteration of the loop, causing significant overheads.

C. Dictionary Entry Selection

Selecting the set of instructions to store in the dictionary per program poses a problem: when the total number of unique instructions in a program increases, a fixed size dictionary can accommodate a smaller portion of them. If the target is minimizing static CR, then selecting entries by their static occurrence is a valid criteria. For minimizing dynamic CR, the most executed instructions should be chosen as dictionary entries. Compressing an instruction occurring only once requires more space than its uncompressed version. In terms of energy consumption, the overhead of programming an instruction to the dictionary should not consume more energy than is saved by executing it as compressed.

D. Bundling

Although allowing uncompressed instructions could be beneficial for energy consumption and execution time, it raises a practical issue of laying out varying width instructions in memory. Fig. 2 compares placement options in a scheme, where instructions have a fixed amount of different templates. If the compressed and uncompressed instruction widths are not divisible by each other, padding bits must be used. As the instruction fetch word size is fixed at design time, the instruction template should minimize or completely eliminate the amount of padding bits.

In Fig. 2a the memory width is aligned to compressed instructions and uncompressed instructions are padded. However, this adds an execution time overhead, as an uncompressed instruction now requires fetching multiple words.

In another approach illustrated in Figures 2b and 2c, the instruction word width is selected so that either an uncompressed instruction or a *bundle* of p compressed instructions can be fetched in one word. This has the advantage of constant number of fetches required for both instruction types. However, it can require padding bits depending on the relationship of uncompressed and compressed instruction sizes. Moreover, a full bundle requires p consecutive compressible instructions. This leads to a design choice: unless p consecutive compressible instructions are found, the bundle is discarded, or the bundle size must be somehow indicated.

IV. Superblock-Based Code Compression

Since the main objective of the proposed method is energy-efficiency, the leading philosophy is to keep the decompression and control hardware simple by moving much of the control of the decompression logic to software. We also expect

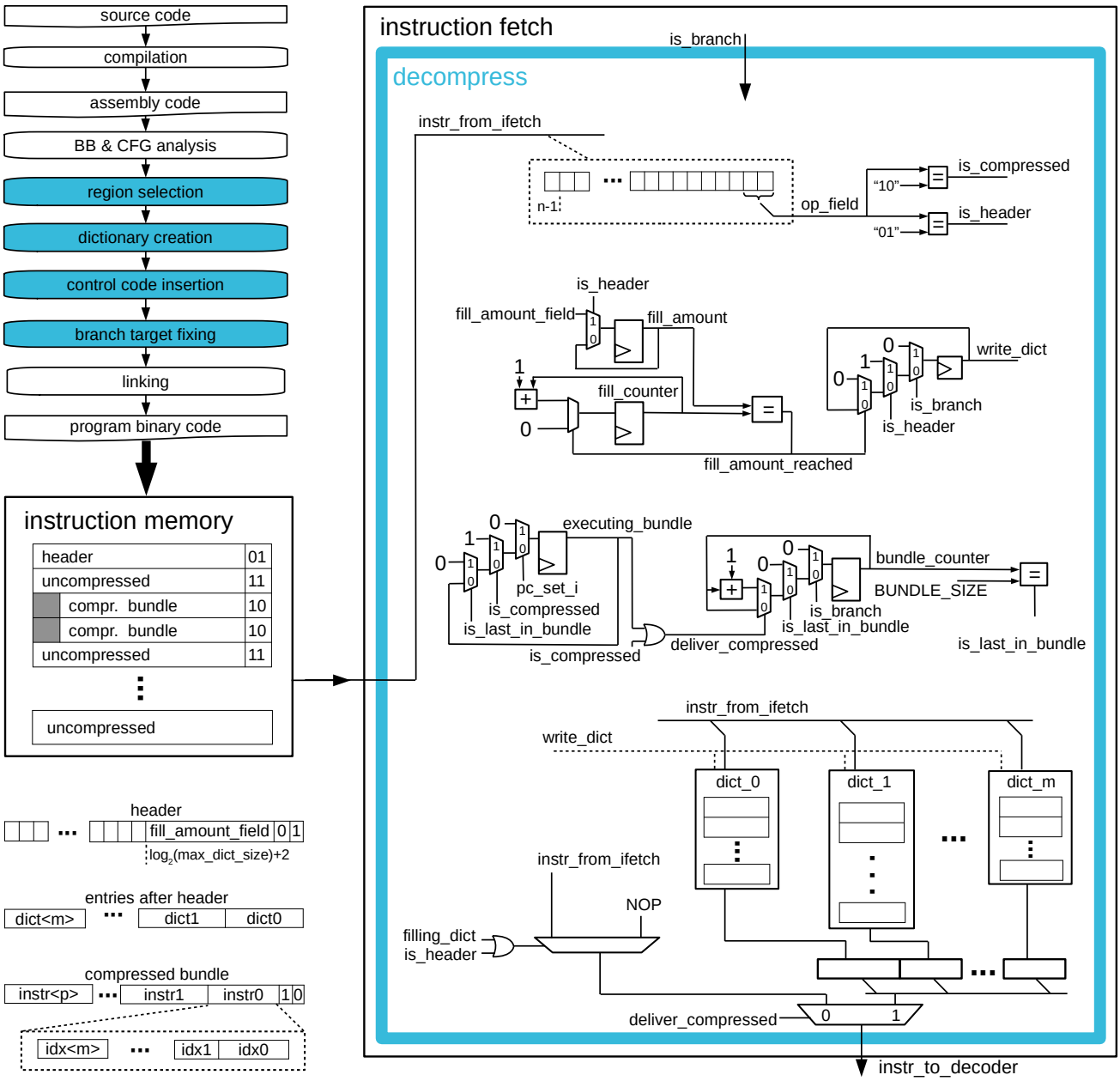
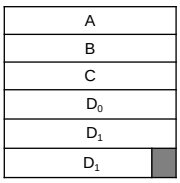
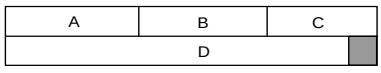


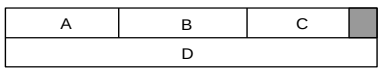
Fig. 1: Overview of the proposed method. Contributions of this work are coloured blue. Dictionary frames created at compile time are programmed into dictionaries. Compressed and uncompressed instructions are read from memory during execution.



(a) Aligned to compressed instruction width.



(b) Aligned to a multiple of compressed instruction width.



(c) Aligned to uncompressed instruction width.

Fig. 2: Alignment of memory width with instruction bundles.

compressing inner loops to provide most benefits, as they are typically program hot spots.

An overview of the proposed method is presented in Fig. 1. At compile time, a CFG is analyzed for each function. From the CFGs, our algorithm analyzes BBs belonging to loops and groups them into compression regions similar to superblocks [16]. For each region, a dictionary frame is created, if it is evaluated as profitable for dynamic CR. This is done with a heuristic algorithm which selects bundles of instructions to compress based on an *occurrence-cost* metric.

During execution, the start of dictionary programming must be indicated. In our implementation, this is a special header instruction that conveys the number of entries to fill. Although it uses an unused encoding in the RISC-V ISA, it does not require modifications to the core itself, as the header is not conveyed

to the instruction decoder, but is handled inside the dictionary decompressor. The header is followed by the dictionary entries to be filled. After a dictionary frame is programmed into the dictionary, compressed and uncompressed instructions can both be executed.

Next, the compile time algorithm used to compress instruction binaries is explained. It is followed by a description of the decompression and dictionary hardware designs.

A. Bundle-Aware Compression Algorithm

As a consequence of allowing uncompressed instructions in program code and concurrently requiring only one fetch per instruction, our approach bundles compressed instructions as shown placed inside the instruction memory in Fig. 1. To keep the fetch control logic straightforward, instructions in the memory are aligned to fixed boundaries according to Fig. 2c. That is, a fetched word can only contain either an uncompressed instruction or a bundle of m compressed instructions. Otherwise, its instructions are left uncompressed. As a result, an instruction can exist both as uncompressed and compressed in different addresses of a program and even inside a BB.

The proposed compression algorithm is described as pseudo code in Fig. 3. First, a CFG is constructed for each function in an input program. From the CFG, nested loops are identified. Correctness of program execution relies on regions being entered only from a single entry point. This allows not having to duplicate the dictionary header code at each location branching into the region. There can be multiple branches out of the region, as the end of a region is not indicated.

As a function call can cause the dictionary frame to be updated, regions with calls in them are not compressed. This is to avoid programming the dictionaries during each iteration of loops, if a function is called from them. Even though all functions are not compressed, we do not assume this during compression, as we perform region analysis per function assuming all functions are not available for a whole program analysis.

We divide full instructions into subfields and compress them individually, as this seems to allow better compression in our preliminary estimations. The algorithm first calculates static occurrences in the region for each instruction subfield defined by the user. For a bundle, the cost of adding new entries is defined as

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

where N is the number of consecutive instructions for a bundle and r_n is the amount of new entries over all subfields in the instruction to be added to the dictionaries to compress the bundle. Next, an *occurrence-cost* $score_{oc}$ is calculated using

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

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.

```
   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 subfield in instruction do
14:                         if subfield.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. 3: Pseudocode to create dictionary frames

For loop regions, the algorithm starts from the innermost loops. Since the algorithm assumes to not have information on loop iteration counts in the general case, all instructions of an inner loop are placed into the dictionary if they fit. This is based on the assumption that inner loop levels are likely to be executed more than outer levels. Upon encountering the first loop level that does not completely fit into the dictionary, that loop is partially placed into the dictionary using the algorithm presented in Fig. 3.

As branch targets can end up in new locations after compression, branch instructions require fixing accordingly. Moreover, branch targets are problematic, as their immediates may have the same encoding as an instruction with another template. For this reason we always reserve an individual entry for each subfield when compressing a branch. Moreover, as RISC-V utilizes PC relative branching, two branches with the same target address cannot use the same entries.

B. Low Overhead Decompression Hardware

While program binaries are compressed at compile time in software, hardware is responsible for decompressing instructions and programming the dictionary entries upon a header instruction. Fig. 1 illustrates the decompression logic used in conjunction with the proposed method. The logic is fairly straightforward, as the complexity of program flow analysis is moved to the compiler. The instruction fetch unit fetches instructions and examines the two least significant bits to

	dict_2	dict_1	dict_0		dict_2	dict_1	dict_0
width	8b	15b	7b	entries	4	64	4

TABLE II: Evaluated dictionary parameters.

check for header and compressed instructions. A header stalls execution until the dictionaries are programmed, after which the execution falls through to the next instruction after the last entry.

C. Design Considerations

For maximizing locality, an option would be to store the dictionary entries directly in consecutive addresses after the header instruction. This allows fast programming of the dictionary as no branching is required to obtain the dictionary entries. However, this is not optimal for loops. Although programming the dictionary is fast in this approach, later iterations of a loop require a branch over the dictionary entries. We locate the dictionary entries after the header instructions, as we assume that a loop region is not entered frequently, keeping the overhead low.

Design parameters for the proposed compressor listed in Table II are manually selected by observing the different instruction templates in the RISC-V ISA [6]. In preliminary evaluations, a bundle size of three instructions provided best results. This translates to a compressed instruction width of 10 bits, as this allows a bundle of three instructions with no padding bits, when the two compression bits are concatenated to a bundle. A large portion of the RISC-V base instruction set opcodes use very similar encodings, requiring few entries for a dictionary. This 7-bit opcode field is then compressed individually. As pointed out by Waterman [19], the majority of immediates used in RISC-V programs are small. Thus, the most significant eight bits of immediates, which are also the most significant bits in the RISC-V immediate encoding, are compressed as one group. The remaining 15 bits are compressed together. Dictionary entry amounts are exhaustively searched from all combinations that result in a compressed width of 10 bits.

V. Evaluation

We evaluate the proposed compression method by integrating it to an embedded system consisting of a processor core and an SRAM-based on-chip instruction memory as illustrated in Fig. 4. A RISC-V implementation called *zero-riscy* [3] is used as the example processor core. Details are listed in Table III. In all evaluations, the 32 kB instruction memory configured in the *zero-riscy* implementation is used. The SRAM energy consumption is estimated using CACTI-P [20].

In order to evaluate the method in different memory hierarchy configurations, we compare against a small direct-mapped *filter cache* [21]. Filter caches are commonly used to improve the instruction locality of data oriented workloads with good energy efficiency due to their relatively simple hardware logic. An open-source implementation [22] by Saljooghi et al. is used for this purpose. The instruction memory access time is set to one clock cycle.

RISC-V ISA	IMC	instruction width	32 bits
issue width	1	pipeline stages	2

TABLE III: Features of the zero-riscy core used in the evaluation.

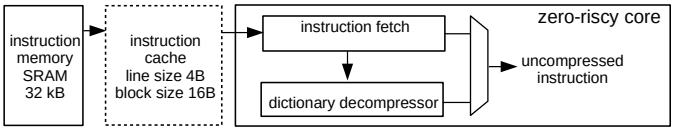


Fig. 4: Evaluation setup.

Benchmark applications from CHStone [23] are compiled using the GCC-based compiler shipped with the *zero-riscy* core. The *jpeg* benchmark is left out of the evaluation as its data does not fit into the 32kB data memory in zero-riscy. Benchmark loop characteristics are shown in Table V. Disassembled program binaries are used as input for the compressor and execution traces from RTL simulation are used to evaluate the dynamic CR and energy consumption.

We compare to the previous state-of-the-art by implementing the algorithm by Thuresson et al. used in the FlexCore processor [7] since it has the finest dictionary programming granularity and is closest to the proposed method. In order to provide another interesting reference point, we compile the benchmarks utilizing the *RISC-V Compressed* (RVC) [6] instruction set extension.

A. Compression Ratios and Runtime Overhead

Static compression ratios are listed in Table VI. On average, RVC reaches a static CR of 0.75, whereas the proposed method results in larger code than the original with a static CR of 1.12 on average. This stems from one of the subfield dictionaries having significantly more entries compared to the two others. In our approach, entries are always programmed to the field dictionaries in parallel. As the entries to be programmed are

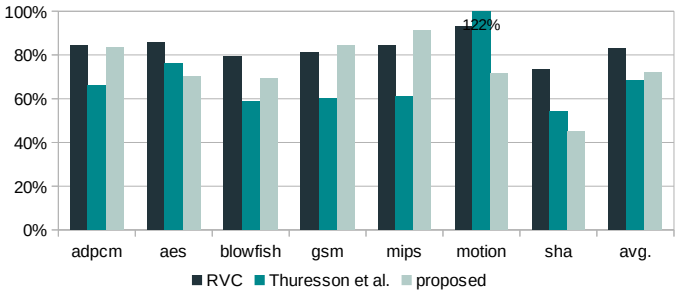


Fig. 5: Dynamic compression ratio. Lower is better.

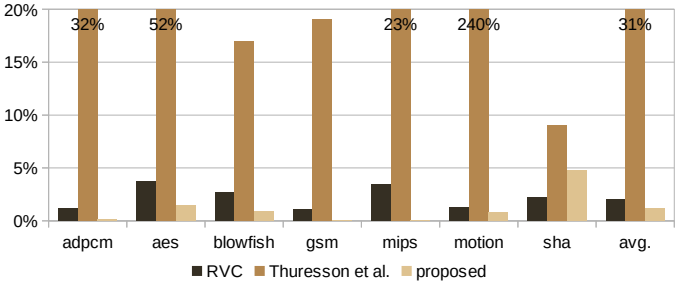


Fig. 6: Runtime overhead.

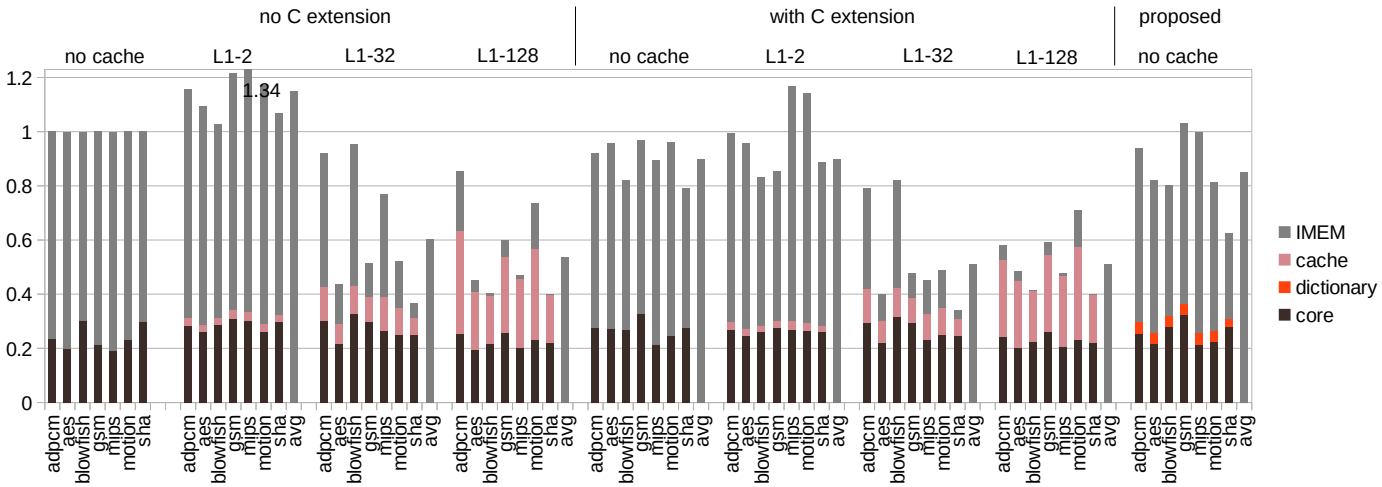


Fig. 7: Energy consumption relative to baseline. Cache evaluated with 2, 32 and 128 lines.

	baseline				RVC				proposed
	no cache	L1-2	L1-32	L1-128	no cache	L1-2	L1-32	L1-128	no cache
adpcm	91502	109.9 %	64.3 %	28.7 %	84.3 %	91.0 %	48.7 %	6.9 %	83.6 %
aes	24698	100.7 %	18.4 %	5.3 %	85.7 %	85.3 %	12.1 %	4.4 %	70.1 %
blowfish	729960	102.7 %	75.2 %	1.5 %	79.4 %	78.9 %	57.1 %	0.2 %	69.4 %
gsm	14586	110.6 %	15.6 %	7.5 %	81.2 %	69.8 %	11.6 %	5.6 %	84.5 %
mips	25420	124.9 %	47.0 %	1.5 %	84.5 %	107.1 %	15.2 %	1.1 %	91.4 %
motion	2202	114.6 %	22.3 %	21.8 %	92.9 %	110.3 %	18.0 %	17.3 %	71.4 %
sha	689790	105.9 %	7.5 %	0.1 %	73.2 %	85.6 %	4.5 %	0.1 %	44.8 %
avg.		109.7 %	27.2 %	3.3 %	82.8 %	88.7 %	17.4 %	1.6 %	72.0 %

TABLE IV: Memory access amounts.

concatenated into a full instruction words and placed into memory, the bit indexes for the smaller dictionaries are left unused after there are no more entries for those dictionaries.

Dynamic CR per benchmark for the evaluated methods is presented in Fig. 5. *RVC* achieves the smallest reduction in fetched dynamic bits, a geometric mean $CR_{dynamic}$ of 0.83. Due to the instruction prefetcher of the *zero-riscy* implementation, there is a small overhead in runtime caused by RVC, as instructions are sometimes fetched during branches.

The algorithm by Thuresson et al. achieves the best $CR_{dynamic}$ with a geometric mean of 0.69. However, the runtime overhead caused by the dynamic dictionary programming is significant: 1.31x compared to uncompressed execution on average. This is due to large loops whose entries do not fit into the dictionaries, forcing costly dictionary programming at each iteration of the loop. The proposed compression achieves on average a $CR_{dynamic}$ of 0.72. This is somewhat higher than that of Thuresson et al., but it should be noted that their method uses dynamic program profiling, whereas our method is based on static CFG analysis and focuses on inner loops to reduce compression bias in data-dependent program flows. As the method of Thuresson et al. compresses all instructions and partially shares dictionary entries between BBs, it achieves better compression ratios than the proposed method. However, the frequent dictionary programming and the requirement to compress all instructions causes a significant execution time overhead, whereas the proposed approach allows uncompressed instructions and hoists dictionary programming code outside of loops, leading to significantly smaller execution time overhead of only 1% on average.

	instructions in loops (%)	instructions executed in loops (%)
adpcm	11.0	47.8
aes	33.4	80.7
blowfish	21.7	90.3
gsm	33.4	80.1
mips	35.8	63.5
motion	16.0	79.8
sha	25.7	93.7

TABLE V: Benchmark characteristics.

	adpcm	aes	blowfish	gsm	mips	motion	sha	avg
RVC	0.78	0.71	0.76	0.74	0.73	0.80	0.76	0.75
Thuresson et al.	1.28	1.25	1.11	1.91	1.82	1.23	1.53	1.42
proposed	1.06	1.17	1.15	1.07	1.10	1.10	1.19	1.12

TABLE VI: Static compression ratio. Lower is better.

B. Energy Consumption

In order to evaluate system energy consumption, we obtain the SRAM instruction memory access energy values from Cacti [20]. Cache numbers and the overhead from the additional hardware are evaluated by implementing the proposed method in SystemVerilog and synthesizing it with the Synopsys Design Compiler using a 28 nm process. As the cache size is small, it is implemented with flip-flops instead of SRAM. Power estimates are obtained for the synthesized design using *switching activity information files* (SAIFs) produced with Modelsim.

As the approach by Thuresson et al. results in a 31% overhead in execution time, we consider this prohibitively large in terms of performance, even though it reaches a slightly

better dynamic CR on average when compared to our proposed method. Thus, we do not implement it in hardware and exclude it from the energy evaluations.

Energy estimates relative to *zero-riscy* with no RVC are presented in Fig. 7. Compared to RVC, the proposed approach reaches on average a 5.5% energy reduction in the instruction stream. The best reduction of 21% is achieved in *sha*.

In *gsm*, the proposed method consumes more energy than the baseline. This is due to a frequently executed loop containing a function call. Our algorithm is not able to compress this loop due to the call, resulting in an overhead from the idling compression logic. This stems again from the program analysis being limited to function CFGs and motivates future research on a whole-program analysis. In *mips*, our current loop detection algorithm is not able to identify the complex main loop structure. Without these cases, the average energy reduction over RVC is 11%.

The zero-riscy core including the decompressor reaches a maximum clock frequency of 1.38 GHz, which is actually 5% better than the baseline, due to the removal of the instruction prefetcher required by RVC. The chip area used by the dictionary decompressor is 4100 μm^2. The latency of the dictionary decompressor component was 0.36 ns with its critical path starting from the dictionary addressing logic and ending in the instruction out port. Critical path of the whole core is not in the decompressor, but in the multiplication unit.

VI. Conclusions

In this paper, we presented a novel programmable code compression scheme to minimize the dynamic compression ratio. We proposed an algorithm to divide an input program CFG into compression regions. As our approach places compressed and uncompressed instructions interleaved in memory, we proposed an algorithm to greedily bundle compressible instructions driven by an occurrence-cost metric.

The proposed compression method was evaluated on a RISC-V processor system with individual dictionaries for instruction subfields. Compared to RVC, our approach reduced instruction stream energy consumption by 5.5% on average and 21% in the best case with only an average 1% runtime overhead.

In the future we plan to investigate whole-program algorithms to more efficiently treat loops with function calls as well as evaluate the method with static multi-issue architectures.

Acknowledgments

The authors would like to thank their funding sources: Tampere University of Technology Graduate School and Academy of Finland (decision #331344). This work is part of the FitOptiVis project [24] funded by the ECSEL Joint Undertaking under grant number 783162.

References

[1] M. Collin and M. Brorsson, "Low power instruction fetch using profiled variable length instructions," in *Proceedings of the IEEE International Systems-on-Chip Conference (SOC)*, 2003.

[2] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *International Symposium on Computer Architecture (ISCA)*, 2011.

[3] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "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)*, 2017.

[4] L. Benini, A. Macii, E. Macii, and M. Poncino, "Selective instruction compression for memory energy reduction in embedded systems," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 1999.

[5] M. Brorsson and M. Collin, "Adaptive and flexible dictionary code compression for embedded applications," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[6] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: Base user-level ISA," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[7] M. Thuresson, M. Själander, and P. Stenstrom, "A flexible code compression scheme using partitioned look-up tables," in *High Performance Embedded Architectures and Compilers (HiPEAC*, Berlin, 2009.

[8] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded risc architecture," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 1992.

[9] C. Lefurgy, P. Bird, I.-C. Chen, T. Mudge, and T. Mudge, "Improving code density using compression techniques," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 1997.

[10] C. H. Lin, Y. Xie, and W. Wolf, "Code compression for VLIW embedded systems using a self-generating table," *Transactions on Very Large Scale Integration Systems*, vol. 15, no. 10, Oct 2007.

[11] S. Seong and P. Mishra, "A bitmask-based code compression technique for embedded systems," in *International Conference on Computer Aided Design (ICCAD)*, 2006.

[12] S. Segars, K. Clarke, and L. Goudge, "Embedded control problems, Thumb, and the ARM7TDMI," *IEEE MICRO*, vol. 15, no. 5, Oct 1995.

[13] K. KISSELL, "MIPS16: High-density MIPS for the embedded market," *Silicon Graphics MIPS Group*, 1997.

[14] C. Lefurgy, E. Piccininni, and T. Mudge, "Evaluation of a high performance code compression method," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 1999.

[15] L. Li, K. Chakrabarty, and N. A. Touba, "Test data compression using dictionaries with selective entries and fixed-length indices," *Transactions on Design Automation of Electronic Systems*, vol. 8, no. 4, Oct. 2003.

[16] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, May 1993.

[17] N. Ishiura and M. Yamaguchi, "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)*, 1997.

[18] W. J. Wang and C. H. Lin, "Code compression for embedded systems using separated dictionaries," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 266–275, 2016.

[19] A. Waterman, *Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed*, 2011.

[20] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design*, Nov. 6-10 2011.

[21] J. Kin, Munish Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 1997.

[22] V. Saljooghi, A. Bardizbanyan, M. Själander, and P. Larsson-Edefors, "Configurable RTL model for level-1 caches," in *Proceedings of the NORCHIP conference*, 2012.

[23] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, Oct. 2009.

[24] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren, and et al., "The FitOptiVis ECSEL project: Highly efficient distributed embedded image/video processing in cyber-physical systems," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2019.