Improving Energy Efficiency of Coarse-Grain Reconfigurable Arrays Through Modulo Schedule Compression/Decompression

HOCHAN LEE and MANSUREH S. MOGHADDAM, Seoul National University
DONGKWAN SUH, Samsung Electronics
BERNHARD EGGER, Seoul National University

Modulo-scheduled course-grain reconfigurable array (CGRA) processors excel at exploiting loop-level parallelism at a high performance per watt ratio. The frequent reconfiguration of the array, however, causes between 25% and 45% of the consumed chip energy to be spent on the instruction memory and fetches therefrom. This article presents a hardware/software codesign methodology for such architectures that is able to reduce both the size required to store the modulo-scheduled loops and the energy consumed by the instruction decode logic. The hardware modifications improve the spatial organization of a CGRA's execution plan by reorganizing the configuration memory into separate partitions based on a statistical analysis of code. A compiler technique optimizes the generated code in the temporal dimension by minimizing the number of signal changes. The optimizations achieve, on average, a reduction in code size of more than 63% and in energy consumed by the instruction decode logic by 70% for a wide variety of application domains. Decompression of the compressed loops can be performed in hardware with no additional latency, rendering the presented method ideal for low-power CGRAs running at high frequencies. The presented technique is orthogonal to dictionary-based compression schemes and can be combined to achieve a further reduction in code size.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; • **Hardware → Power estimation and optimization**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Coarse-grain reconfigurable array, code compression, energy reduction

ACM Reference format:
Hochan Lee, Mansureh S. Moghaddam, Dongkwan Suh, and Bernhard Egger. 2018. Improving Energy Efficiency of Coarse-Grain Reconfigurable Arrays Through Modulo Schedule Compression/Decompression. *ACM Trans. Archit. Code Optim.* 15, 1, Article 1 (March 2018), 26 pages.
https://doi.org/10.1145/3162018

This article is an extension of a paper presented at CGO'17 [10]. The additional material includes a new bin packing–based compression algorithm, a new and more efficient decoder logic, details about the hardware implementation, and new and extended results.

This work was supported in part by Samsung DMC, by BK21 Plus for Pioneers in Innovative Computing funded by the National Research Foundation (NRF) of Korea (grant 21A20151113068), by the Basic Science Research Program through the NRF funded by the Ministry of Science, ICT & Future Planning (grant NRF-2015K1A3A1A14021288), and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

Authors' addresses: H. Lee and B. Egger (corresponding author), Dept. of Computer Science and Engineering, Seoul National University, Gwanak-ro 1, Gwanak-gu, Seoul 08826, Republic of Korea; emails: {hochan, bernhard}@csap.snu.ac.kr; M. S. Moghaddam, Dept. of Electrical and Comp. Engineering, Seoul National University, Gwanak-ro 1, Gwanak-gu, Seoul 08826, Republic of Korea; email: mansureh@dal.snu.ac.kr; D. Suh, Samsung Electronics, Seoul R&D Campus 34, Songchon-gil, Seocho-gu, Seoul 06765, Republic of Korea; email: david1.suh@samsung.com.

https://doi.org/10.1145/3162018

1 INTRODUCTION

The trend for high-resolution displays and high-quality audio on mobile devices has led to a greatly increased demand for solutions that are able to process a high computational load at minimal power consumption. Typical choices for such application domains are field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). Although the latter provide high performance at a relatively low power consumption, the prohibitively high development cost and the limited programmability hinder broad applicability. FPGAs, however, offer high flexibility through bit-level reconfigurability at the expense of a high routing overhead and a long reconfiguration phase. Coarse-grain reconfigurable array (CGRA) processors fill this gap by providing the necessary computational power at a moderate energy consumption while remaining easily programmable [31, 40]. CGRAs have been integrated in several commercially available mobile systems, such as smartphones, tablets, and smart TVs [18, 24, 25, 38].

CGRA processors are built around an array of heterogeneous processing elements (PEs). Data memory, register files, and constant units (CUs) hold and produce values. An interconnection network, composed of multiplexers, latches, and wires, routes data from producers to consumers. Over the past 15 years, many CGRA processors with different architectures and execution modes have been proposed [4, 5, 11, 13, 16, 17, 29, 31, 39, 43, 46]. In this work, we focus on CGRAs that execute modulo-scheduled loop kernels and operate in dataflow mode [5, 31, 46]. The dataflow graph (DFG) of a loop kernel is mapped onto such CGRAs in the form of a *modulo schedule*, a variant of a software-pipelined loop. Compilers generating modulo schedules using simulated annealing [32], edge-based scheduling [36], or, more recently, bimodal scheduling [48] have been proposed. The modulo schedule of a loop kernel contains the configuration bits for every reconfigurable hardware entity for every cycle of the schedule and is stored in decoded form in the configuration memory of the CGRA. This memory is typically implemented as a wide on-chip SRAM. The large number of reconfigurable entities in our target architecture requires memories that are more than 1,000 bits wide and several hundred lines deep, thus occupying a significant amount of the entire chip area. The frequent reads from the configuration memory pose a burden on energy consumption as well, amounting to 25% to 45% of the entire CGRA chip energy consumption [20, 22], although more specialized architectures with a lower energy consumption exist [5].

In this article, we present a hardware/software codesign methodology to significantly reduce the energy spent for reading the modulo schedule from the configuration memory. As a welcome side effect, the size of the generated instruction stream is reduced. The core idea of the presented technique is to identify and eliminate consecutive identical lines from the configuration memory. Duplicated lines do not need to be stored, thereby saving space. At runtime, duplicated lines do not have to be read from the configuration memory, thereby saving energy. Since code generated by standard compilation techniques for CGRAs does not contain a lot of duplicated lines, two optimizations—a spatial optimization implemented in hardware and a temporal optimization applied by the compiler—increase compressibility of the generated code. On the hardware side, the wide configuration memory is reorganized into several independent partitions, each with its own program counter. The partitioning is chosen such that the encoding of hardware entities that are frequently reconfigured in the same clock cycles are grouped together. On the software side, a postpass optimization phase is added to the compiler that minimizes and merges the changes in the encoding to occur in the same clock cycles. At runtime, a simple hardware decoding unit reassembles the configuration lines from the individual partitions. The decoder logic is extremely lightweight and does not introduce additional latency. The optimized code exhibits good compressibility and runtime energy savings. For different application domains with a total of 247 loops, the presented compression scheme achieves, on average, a 63% reduction in the size required to store

the CGRA loop instruction stream and a 70% reduction in energy consumption in the instruction fetch logic.

The presented method has been applied to the compiler and architecture of a commercially available CGRA—the Samsung Reconfigurable Processor (SRP) [46] deployed in smartphones, TVs, printers, and cameras of the same manufacturer [26, 27, 42, 44]. The design has been implemented in Verilog and synthesized using a 45nm manufacturing process.

The remainder of this article is organized as follows. Related work and the organization and operation of CGRAs are discussed in Sections 2 and 3. Section 4 gives an overview of the presented technique and discusses the temporal and spatial code optimization techniques. Section 5 describes the partitioning algorithms, and Section 6 discusses the decoder hardware. Section 7 compares the different partitioning approaches and discusses the results. Section 8 concludes this article.

2 RELATED WORK

Methods for code compression and energy reduction in the instruction decoder for processors and accelerators of embedded systems have been intensively researched over the past decades, ranging from techniques for embedded RISC processors [50] to VLIW processors [9, 14], FPGAs [30], horizontal microcoded architectures [12, 34], and CGRA processors [1, 8, 15, 19, 45].

The classification "coarse-grained reconfigurable array" is used for a wide variety of processors, and their architectures differ significantly. Compression techniques tend to be tailored to the requirements of the target architecture, and, in general, it is difficult to apply a proposed technique without modifications to other types of CGRAs. For example, some CGRA processors allow control flow alterations in the instruction stream [43], whereas others execute code on several parallel lanes similar to GPUs [39] or broadcast instructions over a network [45]. We focus on code compression and energy reduction for CGRAs executing code in the form of modulo-scheduled loop kernels [18, 31].

The parallelism of the architecture, foremostly the number of PEs (PUs), its execution mode, and the complexity of the target applications have a significant effect on the performance of the compression method and are thus difficult to compare. Nevertheless, configuration memory compression methods can be largely divided into three loose and not completely separate categories. Parallelism-based compression methods remove redundant configuration words from the instruction stream. Second are dictionary-based compression techniques that compress frequently used words in the bitstream by replacing them with a more compact version. The third category of compressors uses knowledge about "don't-care bits"—that is, bits whose values do not affect the validity of the computation. Setting these bits to specific values can improve compressibility of the code.

2.1 Parallelism-Based Compression Techniques

Parallelism-based compression techniques are typically applied to CGRAs exploiting data-level parallelism (DLP). The techniques not only eliminate redundancy in the code but also decrease dynamic energy by reducing the number of reads from the configuration memory. For this purpose, these schemes first analyze the pattern of the code. Identical context words that are executed in the same cycle are only stored and fetched only once, which decreases energy consumption. To remove duplicated words, a decompression unit is needed, but the overhead is usually small. Parallelism-based schemes also tend to show good compressibility for specific applications.

MorphoSys [45], one of the earlier CGRA designs, focuses on reducing the energy consumption of the configuration memory by exploiting DLP. Its 64 PEs are organized on an 8x8 array, and one configuration word is broadcast to all eight elements in the selected row/column. This design reduces the necessary memory for the configurations and also amount of data read from it, leading to a reduced energy consumption. RoMultiC [49] extends the concept of row/column

broadcasts by including a multicast bitmap into each configuration word to indicate which PEs are to be reconfigured. To avoid large bitmaps, multilevel bitmaps are proposed where one bit selects multiple rows/columns instead of a single one. The architecture also supports mirroring and folding to further reduce the number of configurations needed. The scheme reduces configuration time by 70% compared to MorphoSys thanks to the reduced number of configurations necessary. An evaluation in terms of configuration size is not provided.

Park and Choi [37] propose a compression scheme for FloRA [23], an 8x8 MIMD CGRA. This architecture supports temporal and spatial code mappings. In temporal mapping mode, the configuration of the i^{th} column is forwarded to the $i + 1^{th}$ column, and only the first column receives a new configuration from the configuration memory. In spatial mode, all 64 elements are configured at once. The authors' scheme removes redundancy in the instruction stream when configuration of the upper four PUs is identical to that of the lower four. The mode (half/full) is encoded in the compressed instructions and decoded at runtime. A compression ratio of 44% is reported.

Although parallelism-based techniques are simple and achieve a significant compression ratio, the main limitation is that the target application code has to be comprised of simple and redundant calculations. Current multimedia codes for CGRAs, however, are often complex and contain only few redundant calculations. In addition, the techniques tend to work well for large CGRAs with lots of idle resources but achieve only minimal code size reductions for smaller arrays and higher code density. Our method also exploits unused components but minimizes the state transitions along the time axis. In addition, components with similar change patterns are grouped into separate partitions to further improve compressibility.

2.2 Dictionary-Based Compression Techniques

Dictionary-based compression techniques represent frequently occurring bitstream patterns with a (shorter) index to a dictionary where the original pattern is stored. During decompression, the original patterns must be fetched from the dictionary and reassembled into the original code. Dictionary-based approaches mainly focus on optimizing the compression ratio of the configuration stream and on minimizing the size of the dictionary. Dictionary-based compression has been applied to RISC, VLIW, FPGA, and CGRA instruction streams.

To minimize the size of the dictionary, Jafri et al. [15] first carried out LZSS compression and then moved on to dictionary compression. They achieved up to 52% of memory reduction. Aslam et al. [1–3] applied state-of-the-art dictionary methods to their large 8x8 CGRA. Similar to the approach presented here, PEs are reorganized to improve compression in the dictionary. The authors report a reduction in code size between 56% and 66%; however, the method only compresses the configuration of the 64 PEs and does not consider other components. In our architecture, the configuration for PEs only takes up 15% of the entire configuration line. Kim et al. [19] proposed a hierarchical dictionary using cache memory. The dictionary is divided into top, medium, and bottom parts. Their technique achieves memory savings of 33%. The benchmarks used for the evaluation are small and may not be good representatives for modern CGRA workloads.

The approach taken by Chung et al. [7, 8] is closest to the method presented in this article. Recent and complex applications can be effectively compressed through their technique because compression is carried out adaptively for each individual target application. The technique exploits spatial and temporal redundancy from the configuration stream and saves the most frequently occurring values in a dictionary. Effective methods for decompression are also described. By taking the decompression overhead into consideration, the authors achieve an average memory reduction of 52% for a variety of benchmarks.

Major concerns of dictionary-based methods are the additional memory requirements needed to store the dictionary and the energy and time overhead of decompression. There is a trade-off

between choosing a big enough dictionary size to allow for good compression while making it small enough so that the additional memory does not consume too much additional energy. In terms of decompression speed, dictionary-based methods typically require a few cycles to reassemble a configuration line. This usually means that the decompressor has to be pipelined or run at a higher frequency than the array. A design of a one-cycle decompression scheme for a dictionary-based compression method is presented by Lekatsas et al. [28].

The method presented in this article is orthogonal to dictionary-based compression schemes. The presented scheme eliminates redundancy in the temporal dimension by cleverly distributing the signals of the entities to different partitions. A dictionary-based scheme eliminates redundancy in the spatial dimension by shortening the length of the encoding. The compression methods of the two schemes are independent and can be combined to yield better compression ratios.

2.3 Don't-Care Bit Compression Schemes

The presented approach improves temporal redundancy for better compression by reducing the number of signal changes over time. This is possible because modulo schedules for CGRAs often contain a significant number of no-operations (`nops`) and other inactive components. Approaches to exploit these don't-care bits for better compression or energy reduction have been proposed foremost for FPGA configuration encoding [12, 30, 34] and VLIW code compression [9] but also play an important role in test vector generation [6]. Murthy and Mishra [34] use graph coloring that excludes don't-care bits from the conflict graph to minimize the number of dictionary entries. Conte et al. [9] encode a "pause" field into VLIW bundles to indicate how many bundles of `nop` instructions follow.

The presented approach uses a two-stage ASAP-ALAN algorithm (see Section 4.3) to fill don't-care bits with the goal to group signal changes of different hardware entities into the same cycle. Compared to previous work [10] where as many signal changes are merged into the same cycle(s) as possible before using an edit distance–based algorithm to split the configuration memory into separate partitions, the work here uses a bin packing–based approach. Hardware entities are added to one of the available partitions based on the result of the ASAP-ALAN algorithm applied to the affected partition only, leading to a significantly improved compression ratio and energy savings.

3 BACKGROUND

The following paragraphs provide the background of modulo-scheduled CGRA architectures and the code generation process targeted in this work.

3.1 Architecture

The computational power of CGRAs is provided by several PEs capable of executing word-level operations on scalar, vector, or floating point data. PEs are often heterogeneous in their architecture and functionality. Several data register files (DRFs) provide temporary and fast storage of data values. Unlike in traditional ISAs, immediate operand values are not encoded directly in an instruction. Instead, CUs are used to generate constant values whenever needed. Figure 1 shows an example of a fictional CGRA modeled after the SRP [46] with 12 PEs, three data and two predicate register files (PRFs), and one CU.

Input operands and results of PEs are routed through an interconnection network composed of physical connections (wires), multiplexers, and latches. This network is typically sparse and irregular. Separate networks can coexist to carry the different data types through the CGRA.

PEs in our architecture support predicated execution—that is, depending on a 1-bit input signal, the PE will either execute the operation or perform a `nop`. PEs can also generate predicate signals

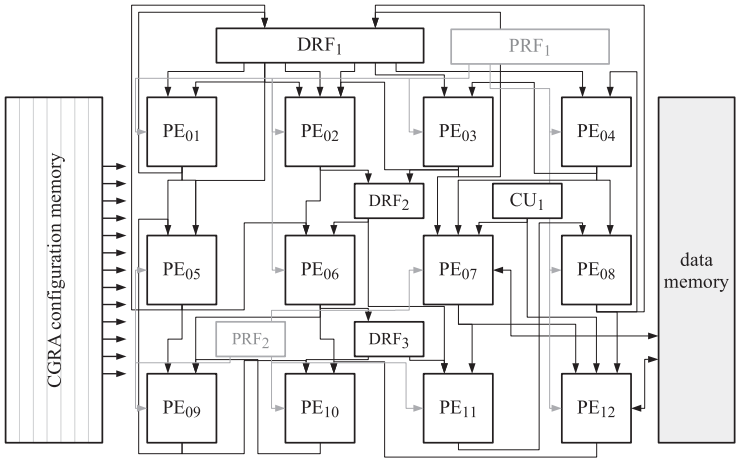

Fig. 1. A coarse-grain reconfigurable array.

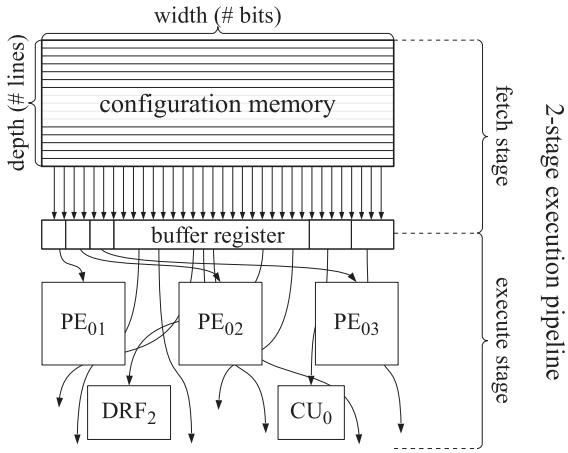

Fig. 2. Configuration memory organization.

that later control execution of operations on other PEs. The PRFs and the separate predicate interconnection network are shown in gray in Figure 1.

3.2 Configuration Memory

The configuration memory stores the execution plan of the CGRA in configuration lines (Figure 2). A *configuration line* represents one cycle in the execution plan in decoded form (i.e., the opcodes for each PE, the register files' write enable and read port signals, the immediate values for CUs, and the selection signal for each of the multiplexers in the interconnection network). For our target architecture, configuration line widths of several hundred bits are the norm—a configuration line of the SRP with 4x4 PEs, for example, is more than 1,200 bits wide. The depth of the configuration is a design parameter and can vary depending on the number and size of loops expected to be executed on the chip. To prevent stalls caused by fetching the loop configuration from off-chip memory, the configuration memory is typically large enough to hold all loops of the running application. The configuration memory of the SRP, for example, is between 128 and 256 lines deep.

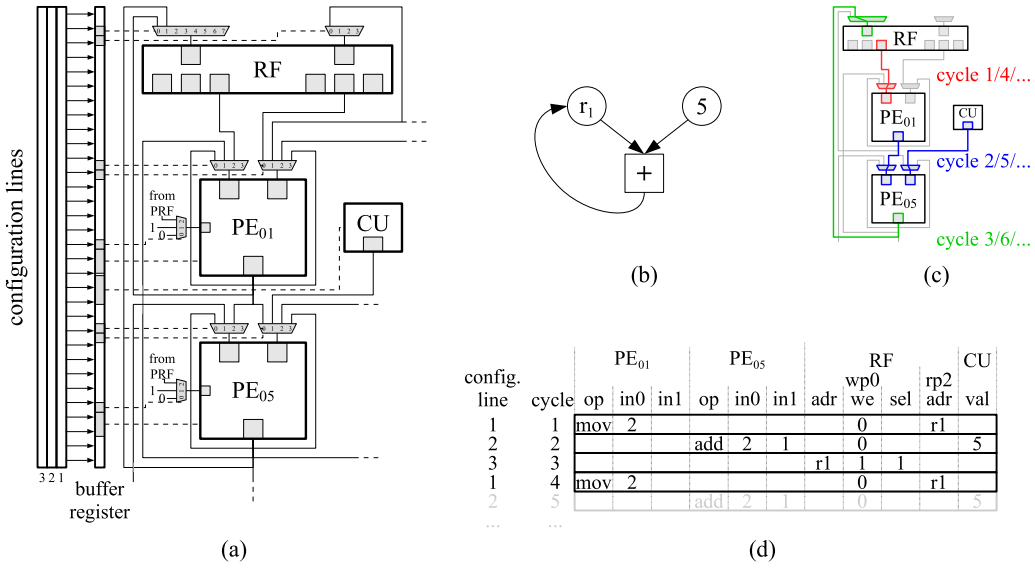

Fig. 3. Encoding a DFG to the configuration lines of a CGRA.

To support high clock frequencies, the configuration of the array is implemented as a two-stage pipeline composed of a fetch and an execute stage. The configuration for the current cycle is held in a buffer register and propagated to the different hardware entities that then perform the requested operation (execute stage), whereas the fetch stage reads the next configuration line from the configuration memory.

3.3 Execution Model

CGRAs targeted in this work execute modulo-scheduled software-pipelineable loops [21, 41]. The kernel of a loop is encoded as several configuration lines and stored in the configuration memory. All entities of the CGRA operate in lock-step, and there is no control flow. A stall caused, for example, by a memory access causes the entire array to stall. The hardware does not provide hazard resolution or out-of-order execution. Similar to VLIW architectures, it is the compiler's responsibility to generate code that does not cause any hazards.

Unlike conventional processors that encode the input/output operands of an operation directly into the instruction encoding and require a decode stage to fetch these operands from the register file or memory, CGRA operations are stored in decoded form. An operation executed on a certain PE at time t will use whatever data is available at the PE's input ports at that time and produce the result of the computation at its output port at time $t + lat$, where lat represents the latency of the operation. If no data is available at the input port, the value of the input operand and consequently the result of the operation are undefined. As an example, consider the CGRA processor in Figure 3(a), showing only two PEs, one register file, and one CU plus parts of the interconnection network. The code to be executed on this array in a loop is $r_1 = r_1 + 5$. The corresponding DFG is shown in Figure 3(b). The constant 5 can only be produced by the CU, which in turn is only connected to PE_{05}. Yet PE_{05} is not directly connected to the register file holding r_1. One possible execution plan is shown in Figure 3(c): load the value of $r1$ into PE_{01} in cycle 1 and forward it to the output port with a latency of one cycle. In cycle 2, PE_{05} selects the output of PE_{01} as operand 1 ($= r_1$) and the output of the CU ($= 5$) as its second input operand, then adds the two.

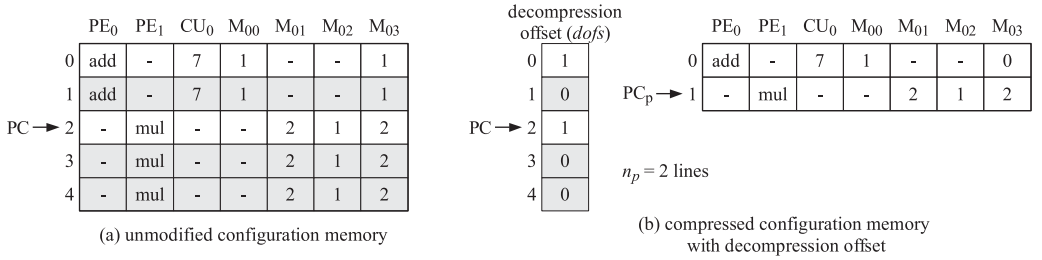

Fig. 4. Deduplication of consecutive identical lines.

The output is produced one cycle later, at time 3, and written back to the register file, and then execution begins anew. The compiler generates all necessary control signals for this code to run as shown in Figure 3(d): register file read/write addresses, register file write enable signals, PE input operand selection at the muxes, PE operation selection, and CU constant generation. `Op`, `in0`, and `in1` of PE_{01} and PE_{05} denote the operation of the PE and the selection signals to the multiplexers at input 0 and 1, respectively. For register file write port 0, labeled `wp0`, the subcomponents `adr`, `we`, and `sel` represent the register address to write, the write enable signal, and the selection signal for the mux in front of the port. For read port 2, labeled `rp2`, the address of the register to be read is given in `adr`. `Val`, finally, is the value to be generated by the `CU`. Empty cells denote entities that are not active in the respective cycle.

3.4 Area and Energy Breakdown

The on-chip SRAM constituting the configuration memory is designed to be large enough to hold all loops of the running application. The configuration memory of CGRAs with an on-chip configuration memory accounts for 10% to 20% of the chip area and consumes 15% to 45% of the total chip energy consumption [5, 20, 22, 35]. The comparatively high energy consumption is due to the fact that a new configuration line is read from the wide configuration memory for every execution cycle of the loop. The presented technique is able to reduce the energy consumption of the instruction memory and decoder logic, on average, by 70% with an area overhead of 8%.

4 COMPRESSION/DECOMPRESSION TECHNIQUE

Modern modulo-scheduled CGRAs operate at clock frequencies from 300Mhz to more than 1GHz, and the cyclewise reconfiguration requires efficient decompressors that can provide a fully decoded configuration line in every cycle. This is difficult particularly for dictionary-based methods, because decoding involves accessing the compressed configuration memory, the memories holding the dictionaries, and shifting the expanded bit patterns into the correct position in the decoded configuration line. Pipelined designs are possible but suffer from overhead in terms of logic and energy consumption. The presented compression scheme was designed under the constraint that code decompression must be possible at the native frequency of the chip with minimal area and energy overhead.

4.1 Compression and Decompression

At the heart of the configuration memory optimization lies a simple compression scheme that eliminates temporal redundancy occurring in the form of duplicated consecutive lines. Figure 4 illustrates the idea. The initial, uncompressed code contains five configuration lines (Figure 4(a)).

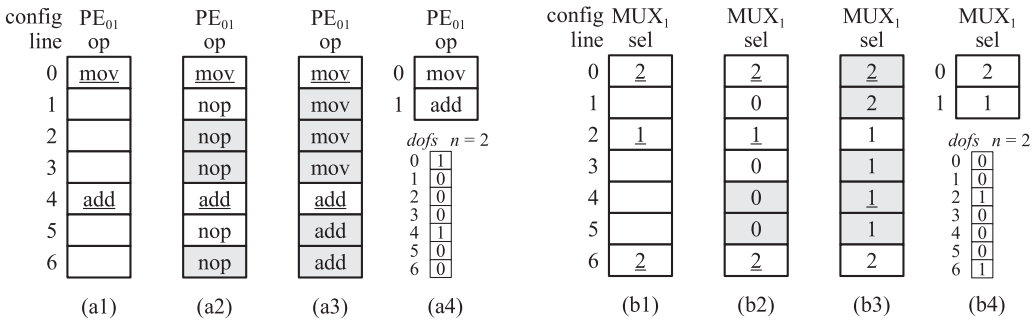

Fig. 5. Increasing temporal locality.

Line 2 is a duplicate of line 1, and lines 4 and 5 are identical to line 3. After compression, only two lines remain, as shown in the right part of Figure 4(b).

Decompression is performed on the compressed configuration using a bit vector denoted *decompression offset* (*dofs*). This vector contains a "1" in all positions of the uncompressed index space where a new configuration becomes active. Zeroes denote that the previous configuration line is repeated. The decompression offset in Figure 4 contains a 1 at index 0 and 2, representing the original indices of the configuration lines. The original program counter *PC* iterates through indices 0 to 4 in the uncompressed code until the loop ends. Decompression requires an additional counter, denoted PC_p. The behavior of program counter *PC* does not change; it keeps iterating through indices 0 to 4 in the decompression offset. PC_p, however, points to the currently active line in the compressed memory. For a given loop index t, $PC_p(t)$ is computed by adding the value of the decompression offset *dofs* to $PC_p(t-1)$. PC_p wraps around when $PC_p(t) = n_p$, where n_p denotes the number of configuration lines in the compressed memory.

$$PC_p(t) = \begin{cases} 0 & \text{if } t = 0 \\ \left\{ PC_p(t-1) + dofs(PC(t)) \right\} \mod n_p & \text{otherwise} \end{cases} \quad (1)$$

For a configuration memory with depth d (see Figure 2), the overhead of the presented decompression scheme in terms of area includes the decompression offset (a $d\,x\,1\,bit$ memory) plus the logic for the program counter for the compressed memory, PC_p. No additional latency is introduced into the fetch stage.

4.2 Increasing the Potential for Duplication

Applied to unoptimized configuration lines, the presented scheme does not achieve a significant level of compression for typical loop kernel code. Experiments with 247 loop kernels from real-world applications for a commercial 4x4 variant of the SRP revealed that in the almost 2,000 configuration lines, there exists not one single line that is identical to its immediate predecessor.

The reasons are twofold. First, a configuration line for the 4x4 SRP architecture is 1,280 bits wide, encoding the configuration signals for 383 distinct hardware entities. It is unlikely that the configuration signals of all 383 entities remain unchanged for two consecutive cycles; this is especially true for software pipelined modulo schedules that are able to hide long latencies by merging code from different loop iterations into the same kernel. The second reason the simple compression scheme does not work is that, in general, schedulers do not generate "compression-friendly" code. Take, for example, PE_{01} in Figure 5(a1) executing a `mov` operation in cycle 0 and an `add` in cycle 4 in a loop with seven configuration cycles. In cycles 1, 2, 3, 5, and 6, the PE is inactive. Current code generators, having to encode a signal for every cycle of the loop, encode `nops` for inactive cycles,

yielding the encoding mov, nop, nop, nop, add, nop, nop, shown in Figure 5(a2). This particular encoding allows elimination of three lines that are identical to their predecessors (grayed-out lines 2, 3, and 6 in the figure). A similar situation exists for the selection signals of multiplexers. MUX_1 in Figure 5(b1) needs to select input 2 in cycle 0, input 1 in cycle 2, and again input 2 in cycle 6 of the loop. During the inactive cycles, code generators typically output a 0 signal. In the resulting code sequence, 2, 0, 1, 0, 0, 0, 2, only two lines (lines 4 and 5) require no signal change and can be optimized away as shown in Figure 5(b2).

Based on these two observations, we present and implement two techniques—a *temporal* and a *spatial* optimization—that greatly increase the likelihood of consecutive duplicated lines in the configuration code for CGRAs. The temporal optimization is implemented into the modulo scheduler of the compiler. It generates more compression-friendly code by keeping the configuration signals of entities constant with the preceding or succeeding line in cycles when the entity is idle. The spatial optimization is a modification of the hardware organization of the configuration memory. The long configuration line is split into several physical partitions that can each be compressed and decompressed individually. The following sections describe these two techniques and their interplay in more detail.

4.3 Temporal Optimization

The temporal optimization is based on two observations about the execution plan stored in the configuration memory. First, in every execution cycle, many of the hardware entities remain inactive. Even when all PEs execute an operation, many of the interconnection network's multiplexers and register file ports are not used and remain unconfigured. Second, data values generated or forwarded by inactive entities have no impact on the correctness of the computation, as these values are never used as inputs to operations of the DFG (excluding operations with side effects such as memory operations). As a consequence, configuration signals of inactive entities can be set to values that improve temporal duplication.

As a motivating example, consider the configuration sequence generated for PE_{01} shown in Figure 5(a1). By propagating signals as long as the entity is idle in the next cycle, the encoding shown in Figure 5(a3) with five compressible lines is obtained. The code is correct because the values produced by the mov/add operations in cycles 1, 2, 3, 5, and 6 are not used as inputs. Figure 5(a4) shows the compressed memory, the decompression offset *dofs*, and the number of lines ($n = 2$).

An interesting situation arises in Figure 5(b1) when the signal of MUX_1 is propagated. Since this code sequence is executed in a loop, we observe that the signal in cycle 0 does not change when wrapping around after cycle 6. The decompressor logic is able to handle this form of duplication. In such cases, if the configuration of the first cycle has been compressed ($dofs(0) = 0$), then the last configuration line must be encoded in the first position of the compressed memory. This is shown in Figure 5(b4) where the selection signal 2 from cycle 6 is encoded in the first position and the encoding of cycle 2 is shifted to the second position. Our definition of PC_p from Equation (1) handles such situations gracefully: $PC_p(0) = 0$, then in cycle 2, $PC_p(2) = 1$ because $dofs(2) = 1$. In cycle 6, PC_p is incremented to two, but the modulo condition with $n = 2$ makes sure that PC_p wraps around and reads the required signal from the first position in the compressed memory.

4.4 Achieving Maximum Compressibility for Multiple Entities

Configuration lines, even when partitioned, comprise the configuration signals of several entities, not just one. The simple downward propagation from the previous section will not lead to optimal results: to maximize compressibility, the signal changes of the individual entities should occur in the same line(s). We have developed a two-step algorithm that optimizes standard modulo

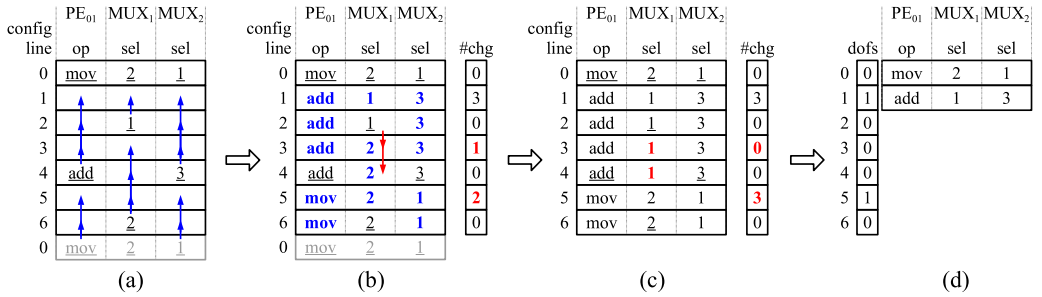

Fig. 6. The ASAP-ALAN propagation algorithm.

schedules containing only configuration signals representing the DFG (i.e., entries of unused entities are empty). In the first step, the configurations of each entity are propagated upward (backward in time) as far up as possible to the previous configuration of the same entity (as-soon-as-possible (ASAP) step). The algorithm respects the modulo time m of the schedule (i.e., the propagation of signals through cycle 0 continues at cycle $m-1$). Figure 6(a) and (b) show the initial modulo schedule before and after ASAP propagation (blue arrows). We keep track of the number of signal changes per line with respect to the preceding line in the #chg vector. If the number of configuration changes per line is zero, the entire line can be eliminated (indices 0, 2, 4, and 6 in Figure 6(b)).

Maximal compression is achieved if the number of changes per line is either zero (line can be eliminated) or equal to the number of entities in the line (all signals change at the same time). The changes per line vector in Figure 6(b) contains two lines that are not optimal: line 3 with one change and line 5 with two changes. If the entity causing the change remains constant from the first to the second line and is inactive in all cycles, the configuration of the line preceding the first suboptimal line can be propagated down to the second one. This step is called *as-late-as-necessary* (ALAN) *propagation* (red arrows). Consider $\mathtt{MUX}_1$. It causes the only signal change in line 3 but remains constant until and including the next line with more than zero configuration changes (line 5). The ALAN step propagates the signal of line 2 down to line 4, causing $\mathtt{MUX}_1$ to switch in line 5. This transformation is shown in Figure 6(c). The result is optimal, as the number of changes is either zero or equal to the number of entities for all configuration lines. The compressed code with the corresponding decompression offset are shown in Figure 6(d). In effect, the algorithm groups signal changes into as few lines as possible, leading to a better compressibility of the code.

One concern is that replacing nops of PEs with ALU operations increases the dynamic energy consumption of the array. Execution of idle non-nops is prevented by setting the predicated execution bit of the PE to zero, effectively blocking execution of the operation (see Figure 3(a)). This technique also takes care of operations with side effects and reduces the switching activity in the array, thus leading to a reduced power consumption in the array.

4.5 Spatial Optimization

The more hardware entities are encoded into one configuration line, the less likely it is that the presented temporal optimization finds enough movable slots to group signal changes into fewer lines. The spatial optimization exploits this fact. It improves compressibility by splitting the configuration line into several *partitions*. Each partition is then compressed separately. Figure 7 shows an example with four hardware entities grouped into one configuration line. The code after the temporal optimization is shown in the upper part and the corresponding compressed configuration along with the decompression offset in the lower part of Figure 7(a). Assuming that each signal of the four

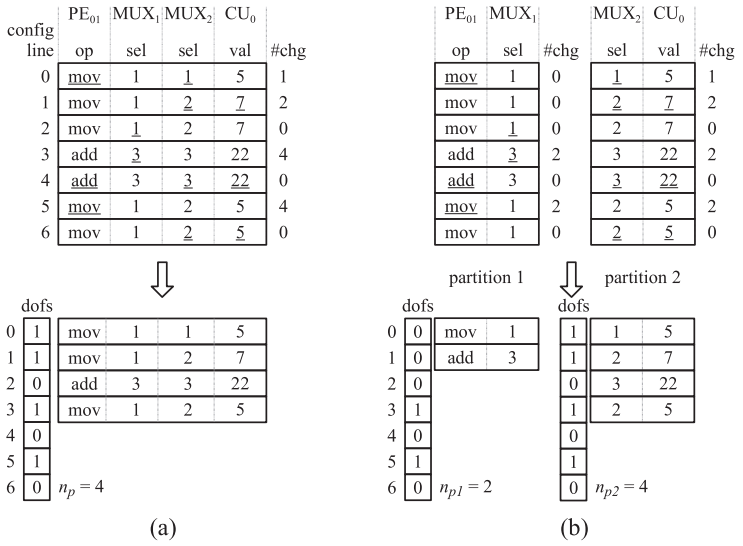

Fig. 7. Improving compressibility by partitioning.

entities occupies 8 bits, the size of the configuration memory is reduced from 7 lines $*$ 4 entities $*$ 8 bits $=$ 224 bits to $4 * 4 * 8 + 1 * 7 = 135$ bits. The $1 * 7$ term represents the space required to store the decompression offset. By splitting the four-entity-wide lines into two partitions each containing two entities as shown in Figure 7(b), the first partition can be compressed with only two lines, whereas the second partitions still requires four. Thanks to this separation, however, the total size of the compressed configuration shrinks from 135 to $2 * 2 * 8 + 4 * 2 * 8 + 2 * 7 = 110$ bits. Note that with two partitions, $2 * 7 = 14$ bits are required to store the decompression offset.

Whereas the temporal optimization presented in the previous sections is applied to code at compile time, the spatial optimization requires modifications to the hardware of the CGRA and is thus a static optimization. The partitions are computed at design time of the CGRA and cannot be modified thereafter. Typically, CGRAs run more than one application, and a partitioning should thus produce good results for a variety of applications. In addition, partitioning introduces a minimal overhead. Since each partition is compressed individually, a separate decompression offset table needs to be generated and stored for each partition. At runtime, additional program counters are required to enable independent reads from the different partitions. The next section describes how to generate a good partitioning while considering several applications. A hardware implementation with an analysis of area and power consumption is provided in Section 6.

5 PARTITIONING BASED ON STATISTICAL ANALYSIS

The compression technique described in the previous section achieves good compression ratios when applied to individual loops. The average compression ratio for the 247 loop kernels from real-world applications is below 0.2 (i.e., the technique reduces the memory requirements by more than 80% with only four partitions). In reality, however, CGRA chips execute not only one but several applications, each comprising one or several loop kernels. The presented compression technique is static in the sense that once the partitioning has been determined at design time of the chip, the compiler has to adhere to the predetermined partitioning scheme and encode each hardware entity into its predetermined partition.

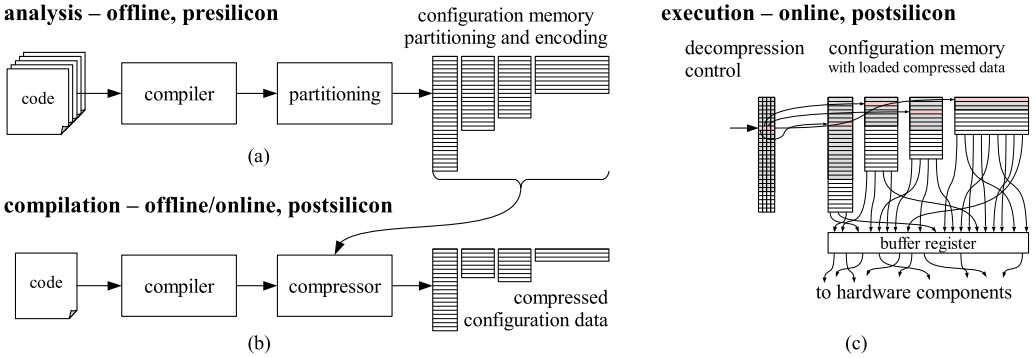

Fig. 8. Analysis, compilation, and execution.

Figure 8 illustrates the three distinct phases of the presented technique: (a) analysis and partitioning, (b) compilation and compression, and (c) decompression and execution. In the *analysis* phase, loop kernels are first compiled for the CGRA. A partitioning algorithm then computes a partitioning that maximizes compressibility for the given loops based on a statistical analysis of the code. The analysis and partitioning is performed *presilicon* as part of the optimization process of a CGRA to a certain application domain. Two techniques are implemented and compared: a scheme based on an edit distance heuristics introduced in previous work [10] and a new greedy algorithm based on bin packing. At *compile time*, the compiler takes a given partitioning as an extra input. It encodes the configuration signals of the different hardware entities into the predetermined partitions and finally applies the temporal optimization described in Section 4.3. Finally, the individual partitions are compressed by removing consecutive duplicated lines, and the decompression offsets are generated for each partition. During *execution*, the compressed code and the decompression offsets are first loaded from the application binary into the partitioned configuration memory, then executed on the CGRA.

5.1 Configuration Memory Partitioning

Computing an optimal partitioning for given loop kernels is a variant of the bin packing algorithm, a combinatorial NP-hard problem. We present and compare two heuristics to generate a partitioning for a given number of partitions and configuration lines. The first heuristic is based on the edit distance between the signal changes of different hardware entities. The second heuristic is a greedy bin packing algorithm. The following sections describe the two algorithms in detail.

The input is the number of partitions n and the configuration lines of a (set of) loop(s). The output is a partitioning of the configuration line into up to n partitions. A partitioning is a mapping $f : E \rightarrow P$, where E denotes the set of configurable hardware entities and P the set of partitions.

5.2 Edit Distance–Based Memory Partitioning

The edit distance denotes the number of editing operations required when transforming one string into another. The smaller the edit distance, the more similar are the two strings. Applied to the signal change patterns of individual hardware entities, a small edit distance implies that many of the signal changes of the entities occur in the same cycles. The edit distance–based heuristic, as presented in prior work [10], first applies the ASAP-ALAN algorithm to the uncompressed configuration lines. Then an ordering of the encodings of the individual hardware entities is generated that groups encodings with similar signal change patterns based on edit distance. Finally, the

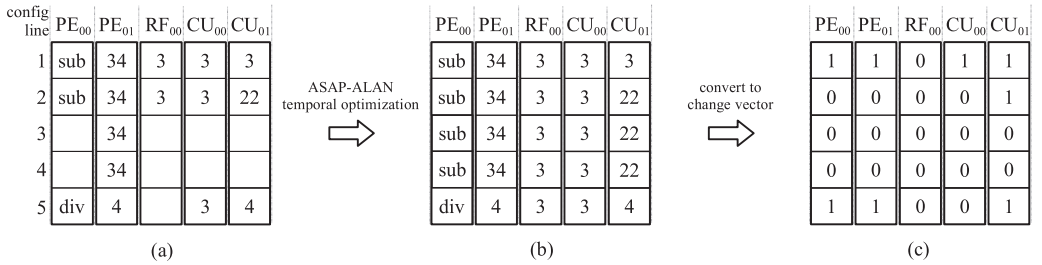

Fig. 9. Converting a configuration into change vectors.

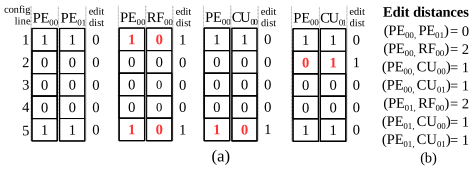

Fig. 10. Computing the edit distance.

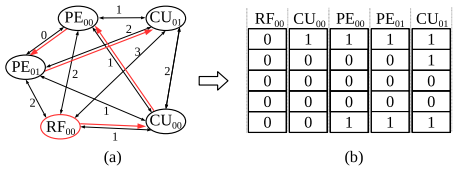

Fig. 11. Ordering hardware entities.

ordered list of configuration signals is traversed and the configuration line split into up to n partitions in a way that maximizes compressibility of partitions.

The temporal optimization is executed as outlined in Section 4.3. To generate an ordering, for each individual entity $e \in E$, the sequence of configuration signals is first converted into a bit vector denoted *change vector*. A "1" at position t implies that the configuration value has changed from the previous cycle $t-1$ to the current one; a "0" denotes that the configuration remains identical. Figure 9 illustrates the process. The change vectors are sorted by the edit distance between the unified change vector of all sorted entities and the change vector of the entity to be added. We use an augmented edit distance that is the sum of the edit distance plus the difference in the number of bits in the unified change vector before and after adding the candidate vector. Intuitively, the additional component allows us to distinguish between vectors that have the same editing distance but cause a different number of configuration line additions as illustrated in Figure 10. To generate an ordering, conceptually a fully connected bidirectional graph, G=(V, E), is generated where V denotes the set of hardware entities and E represents the edges between the entities. The weights on the edges are set to the augmented edit distance between the two components. An ordering is generated by starting with the most invariant entity (i.e., minimal number of ones in the entity's change vector), then the graph is traversed along the edges with the smallest weights until all nodes have been visited exactly once. If several edges have the same minimal weight, one is chosen randomly. Figure 11 illustrates the process for the running example. The resulting order has entities with similar signal change patterns located closer to each other.

In the last step, the cut-off positions that divide the configuration line into up to $n-1$ partitions are identified in a greedy manner. The sorted list of entities is scanned from the first to the last position. In each step, the memory savings are computed by multiplying the bitwidth of the included entities by the number of zeros in the unified change vector. If the inclusion of the next change vector reduces the amount of memory that can be saved, the position is marked as a cut-off point. Figure 12(a) illustrates how the two cut-off points are chosen (for the example, an encoding width of 1 bit for all entities is assumed). The resulting partitioning, shown in Figure 12(b), achieves memory savings of 16 bits or 64%.

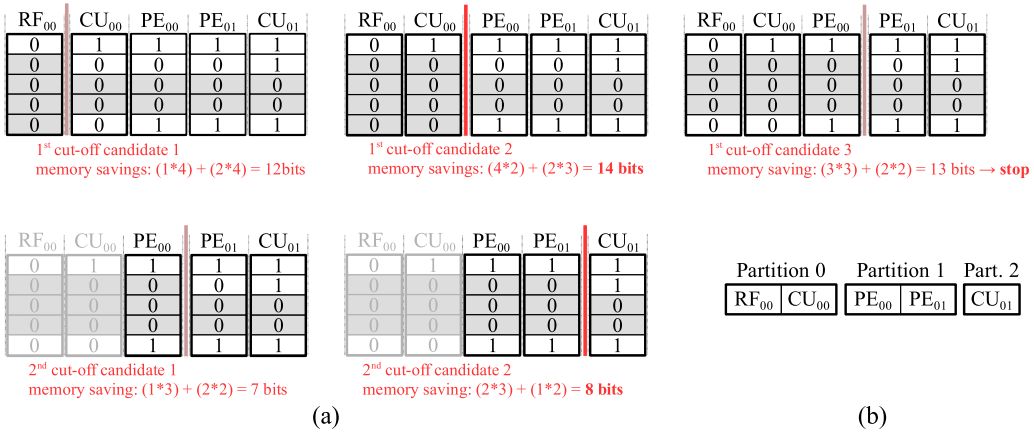

Fig. 12. Edit distance–based partitioning: selecting the cut-off point for three partitions.

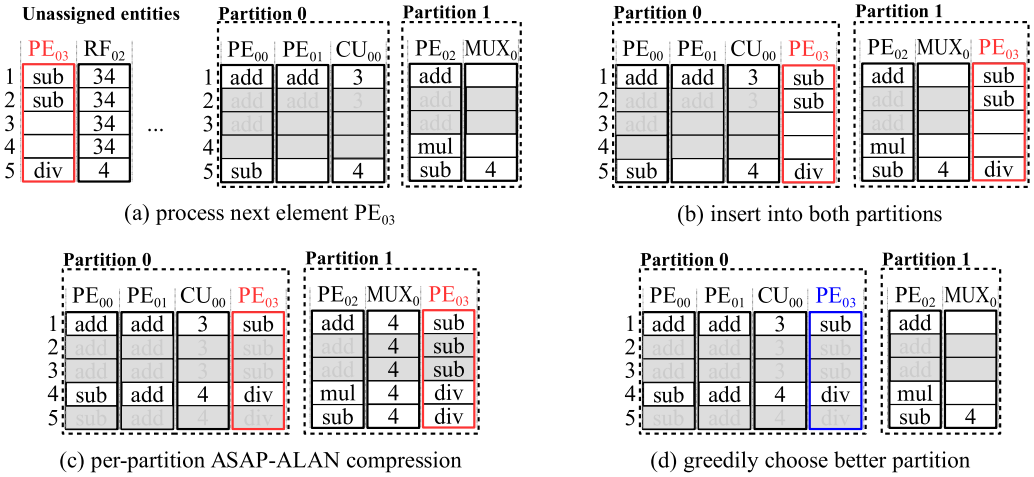

Fig. 13. Packing process of bin packing compression.

A problem with edit distance–based partitioning is that the ASAP-ALAN temporal optimization is computed once at the beginning on the entire, uncompressed configuration lines. Especially the ALAN step has more opportunities to merge signal changes if the number of entities is smaller. In the following bin packing–based memory partitioning scheme, this deficiency is eliminated.

5.3 Bin Packing–Based Memory Partitioning

The bin packing–based partitioning scheme first creates the desired number of empty partitions (the bins). It then packs the hardware entities one by one into the existing bins. The partition for a given entity is determined by temporarily inserting the entity into all bins, then running the ASAP-ALAN optimization algorithm separately on each bin and computing the expected memory savings. The bin that achieves the best memory savings is selected as the target partition.

One step of this bin packing process is illustrated in Figure 13. Figure 13(a) shows the existing two bins and the list of yet to be inserted entities. Entity PE_{03} is inserted into both bins in Figure 13(b), and the result of applying the ASAP-ALAN optimization individually to both partitions

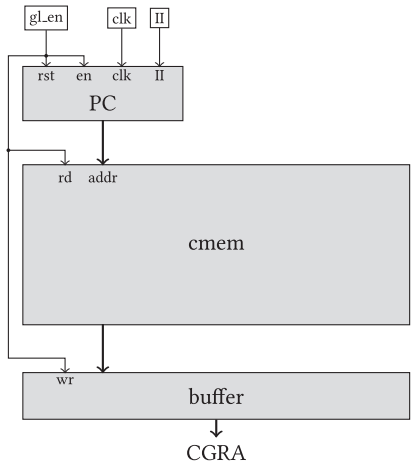

Fig. 14. SRP [46] configuration memory.

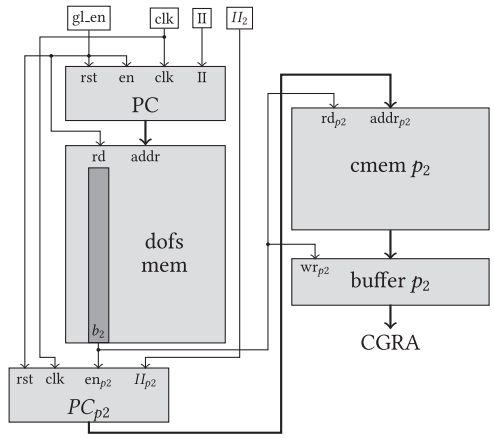

Fig. 15. Presented partitioned configuration memory; details are only given for the second partition (P_2).

is shown in Figure 13(c). There were three compressible lines in partition 0 before PE_{03} was inserted, yielding 9 bits of memory savings (assuming that all entities are 1 bit wide). Inserting PE_{03} changes the timing of the signal change from 5 to 4, but still three lines can be eliminated resulting in 12 bits or a relative change of +3 bits of memory savings. For partition 1, the savings are 4 bits before and 6 bits after inserting PE_{03} (i.e., additional savings of +2 bits). The improvement when inserting the entity into partition 0 is higher than that for partition 1, so partition 0 is chosen, as shown in Figure 13(d).

An advantage of bin packing–based partitioning is that the desired (maximal) widths for partitions can be given as an additional constraint if required by the hardware design of the chip.

6 HARDWARE DECODER LOGIC

The unmodified instruction fetch logic comprises a program counter *PC*, the configuration memory holding the execution plan of the loop, and a buffer register. The buffer register acts as a pipeline register dividing execution of a configuration line into a fetch stage in which the *PC* is used to load the next configuration line from the configuration memory and an execution stage that executes the configuration stored in the buffer register (see Figure 2). Figure 14 shows a block diagram of the hardware components involved in fetching instructions. The *global enable* signal *gl_en* is high during execution of a loop. The initiation interval *II* defines how many configuration lines the loop body encompasses, and the program counter repeatedly issues the line addresses $0, 1, \ldots, II-1, 0, 1, \ldots$ until *gl_en* goes low. The enable, read, and write signals *en*, *rd*, and *wr* are tied to *gl_en* (i.e., are always active for the entire duration of the loop). This means that for every cycle, a configuration line is read from the configuration memory and copied into the buffer register before computing the updated program counter.

Figure 15 shows the high-level organization of the presented partitioned configuration memory architecture. The decompression offset memory *dofs mem* holds the decompression offsets of the individual partitions. The configuration memory is divided into several partitions *cmem* p_i, each with its own program counter PC_i. The buffer register is also split to match the partitions (*buffer* p_i); this is necessary because not all partitions write to the buffer register in every cycle

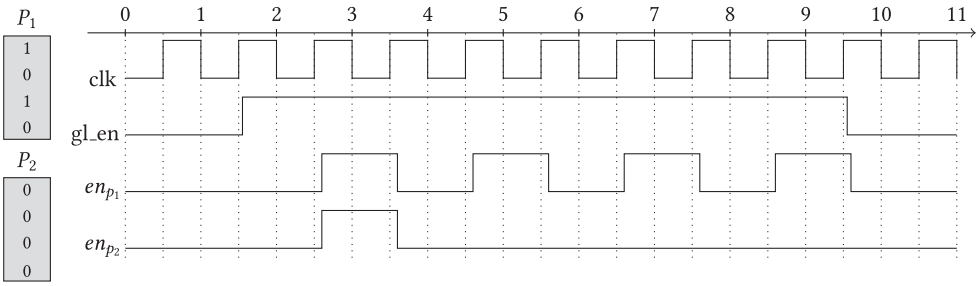

Fig. 16. Illustration of two iterations of a loop kernel with two partitions: II_1=2 and II_2=1.

(i.e., individual write signals are necessary). The program counters for the individual partitions operate as defined by Equation (1). The number of configuration lines per partition is denoted II_i to keep the naming consistent.

The decoder logic has been designed for optimal energy savings. The $dofs_i(t)$ for a partition i, indicating whether a new configuration is becoming active or not, controls the enable signal of the partitions' program counters, which can then be implemented as simple 1-adders. In addition, the value of $dofs_i(t)$ also controls the read/write signals of the corresponding configuration memory partition *cmem* p_i and the buffer register *buffer* p_i. In other words, the partitions of the partitioned configuration memory are only read when a new configuration line is fetched from the memory, resulting in a lower dynamic (read) energy consumption.

Figure 16 shows the control signals for the execution of a loop for two partitions: p_0 and p_2. In cycle 2, *gl_en* goes high, causing the array to enter the loop. In the first cycle after entering a loop, the very first configuration line has to be fetched from all partitions, and hence the enable signals for the two partitions, en_{p1} and en_{p2}, go high; rd_{pi} and wr_{pi} are equal to en_{pi} (not shown in the figure). After the first cycle, the enable signals are controlled by the decompression offsets for the individual partitions shown in the left lower corner of the figure. For every four cycles, partition 1 is read twice. Partition 2 shows an interesting optimization: if a partition contains only exactly one configuration line, the decompression offset is set to all zero. This will cause the decoder logic to read the configuration line exactly once when entering the loop and never after.

7 EVALUATION

7.1 Experimental Setup

The presented method is evaluated on a commercial CGRA: the SRP [46]. The processor consists of 16 PEs, 12 register files, 8 CUs, and a large number of multiplexers. A configuration line for the total 383 configurable entities is 1,280 bits wide. The ASAP-ALAN compression scheme and the two partitioning methods have been implemented in the proprietary Samsung C Compiler for CGRAs. The overhead of the partitioned configuration memory and the necessary decoder logic in terms of area, power, and timing information has been synthesized for a 45nm manufacturing process with the Synopsis Design Compiler [47]. For the configuration memory energy computation, CACTI 6.5 [33] is used.

7.2 Benchmarks

The benchmarks used for the evaluation are 32 real-world applications deployed in smartphones, cameras, printers, and other high-end mobile devices manufactured by Samsung. The applications contain a total of 247 loop kernels and 1,978 configuration lines, and the result of considering all 247 loop kernels at once is labeled `All loops`. CGRAs often target a specific application domain

Table 1. Application Domains With Applications and Number of Loop Kernels per Application

Application Class	Applications (# kernels)	Total Kernels	Total Conf. Lines	IPC
Face Detection	face_detection (35)	35	243	7.4
Graphics	3D (9), matrix (3), opengles_r269 (9), PICKLE_V1.2 (14)	35	155	3.4
Imaging	csc (1), dct (1), median (1), sad (1), gaussian_filter (19)	23	391	7.8
Resolution	bilateral (3), gaussian_smoothing (3), optical_transfer_function (6)	12	101	7.3
Video	aac.1 (16), avc.swo (11), mp3.1 (10), EaacPlus.1 (23), mpeg_surround (26)	86	588	3.3
Voice	FIR (1), huffman_decode (6), bit_conversion (1), histogram (1), amr-wbPlus (12)	21	174	3.2
Voice (SIMD)	FIR (1), high_pass_filter (1), huffman_decode (5), bit_conversion (1), histogram (1)	9	116	3.5
Others	word_count (13), merge_sort (5), bubble_sort (3), array_add (5)	26	210	1.0
All loops	All 32 applications	247	1,978	4.6

at design time; to reflect this, applications are grouped into eight application domains. For the `Voice` application domain, two different benchmark sets exist: one using only general-purpose operations and the other optimized for SIMD operations. Table 1 lists the application domains, benchmark applications, and average instructions per cycle (IPC) for each domain.

7.3 Overhead Versus Benefit

The more partitions the configuration is split into, the more flexibility the compression algorithm has, and thus a higher memory reduction can be expected for a larger number of partitions. Partitioning, however, comes with two kinds of overheads: area overhead and the consumed energy in the decoder logic. In the following, let n denote the number of partitions.

The area and energy overhead is caused by the additional hardware logic for the program counters and the n bit wide memory for the decompression offset. The *dofs* memory and the partitions are accessed individually and thus need to be composed of physically distinct memories. In the SRP, the configuration memory is composed of 64, 32, and 16 bit wide memories; the same building blocks are used to compose the partitions. Smaller memories have a lower bit density per area because of the access logic required for each memory. In addition, since the smallest memory is 16 bits wide, each partition can have up to 15 bits of padding. A 70 bit wide partition, for example, would be composed of one 64-bit and one 16-bit memory block, leading to a padding of 10 bits. For each partition, a separate program counter PC_p is required.

Figure 17 shows (a) the memory reduction, (b) the area overhead, and (c) the energy breakdown normalized to the unmodified architecture in dependence of the number of partitions on a logarithmic scale. The results are reported for `All loops` and use the bin packing–based compression scheme. We observe that compressibility increases with an increasing number of partitions

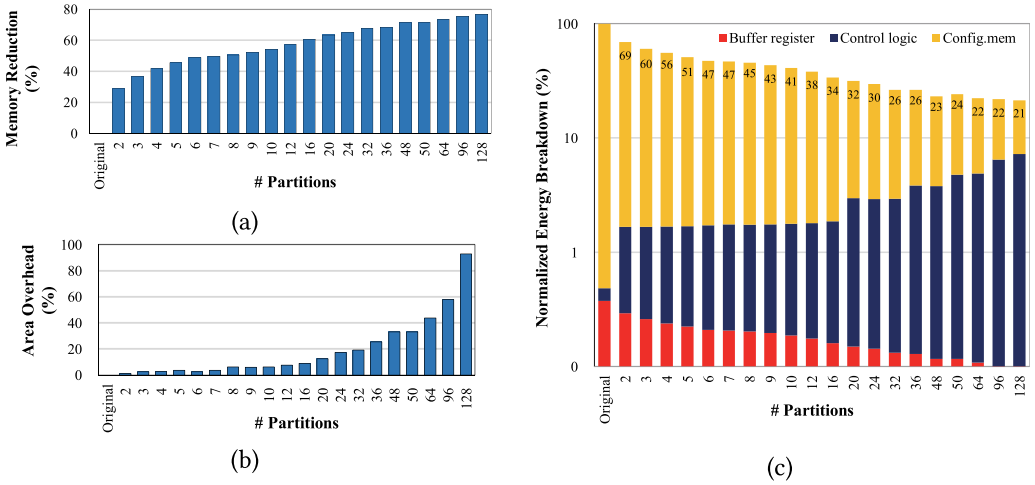

Fig. 17. Memory reduction, area overhead, and energy breakdown in dependence of the number of partitions.

Table 2. Memory and Energy Reduction for Edit Distance– and Bin Packing–Based Partitioning With 16 Partitions

Application Classes	Edit Distance			Bin Packing		
	Memory Reduction (%)	Runtime Energy Reduction (%)	CPU Time (sec)	Memory Reduction (%)	Runtime Energy Reduction (%)	CPU Time (min)
All loops	48	45	134	61	66	942
Face Detection	46	42	18.8	55	63	40.4
Graphics	60	56	13.7	67	83	27.0
Imaging	39	36	24.6	49	49	57.0
Resolution	41	38	8.3	52	56	14.3
Video	61	58	43.0	73	81	141
Voice	61	58	12.6	72	79	26.7
Voice (SIMD)	64	61	8.2	76	79	14.8
Arith. Mean	52.5	49.3	32.9	63.1	69.5	157.9

from 0% for a single configuration memory to 76% for 128 partitions. The area overhead is moderate up to about 20 partitions with an overhead of 12% but then quickly raises to reach 93% for 128 partitions. Padding in the partitions amounts to 128 bits for 20 partitions and reaches 992 bits for 128 partitions. The energy breakdown in Figure 17(c) reveals that the total energy consumption spent in the instruction fetch logic falls as low as 21% for 128 partitions even though the overhead of the decoder logic reaches 7.1% (a 64-fold increase from 0.11% in the original design). This is thanks to the reduced number of reads from the partitions and writes to the buffer register. The decompression offset requires one 16-bit memory up to 16 partitions, 2 up to 32, and so on; this composition is clearly visible in the energy overhead of the control logic.

7.4 Edit Distance– Versus Bin Packing–Based Partitioning

Table 2 compares edit distance–based compression [10] with the bin packing–based algorithm presented in this work for 16 partitions each and the different application classes.

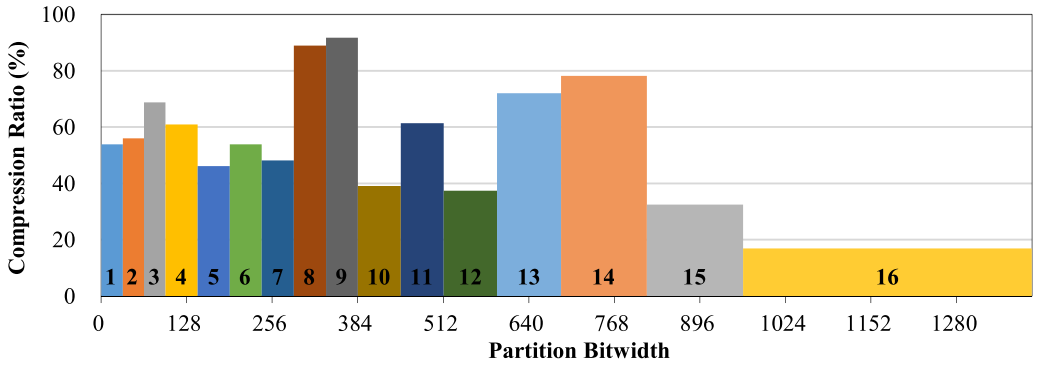

Fig. 18. Bitwidths and compression ratios for the bin packed partitions of the `Face Detection` domain.

The bin packing–based algorithm consistently outperforms edit distance–based partitioning for compressibility and runtime energy reduction. On average over all application classes, bin packing achieves a 10.6% higher compressibility and a 20% better runtime energy reduction. The energy reduction is comparatively higher for bin packing, as this algorithm can generate more partitions with only a single configuration line, which at runtime are read exactly once per loop execution, as discussed at the end of Section 6.

In terms of execution speed, on average, the edit distance–based algorithm with linear complexity is several orders of magnitude faster than the bin packing–based one; however, the latter achieves a significantly higher memory and runtime energy reduction. It is important to note that the computational overhead is only incurred once at chip design time; once the partitions have been computed, compilation speed is not affected. In light of this, the presented bin packing–based algorithm clearly outperforms the edit distance–based version.

7.5 Analysis of Compression and Energy Reduction

Figure 18 visualizes the results of partitioning and compressing code of the benchmarks from the `Face Detection` application domain. The *y*-axis shows the compression ratio (lower is better). The bin packing algorithm has packed entities with low compressibility into several small partitions. Entities with low activity are packed into one large partition (partition 16) with a compression ratio as low as 17%.

From Table 2, we observe that there is a significant difference in memory and runtime energy reduction in dependence of the application domain. For bin packing, the `Imaging` application domain shows the lowest numbers with a 49% reduction in both memory and energy consumption, whereas the `Voice (SIMD)` domain shows excellent compressibility and energy savings of 76% and 79%, respectively. The presented method exploits unused configurations (don't-care bits) in the instruction stream; application domains with a higher utilization of the hardware entities are thus expected to perform worse than those with a lot of unused entities. The IPC measure, shown in the last column of Table 1, is an indicator of the overall utilization of hardware entities. Indeed, a clear correlation between IPC and compressibility can be observed: `Imaging` has the highest IPC of all application domains with 7.8 and `Voice (SIMD)` one of the lowest (3.5). Figure 19 visualizes the relationship between IPC and compression ratio for all kernels of the different application domains. Circles represent individual kernels, and the diamond-shaped boxes show the average IPC and compression ratio of application domains.

This result suggests that the presented method may not yield a significant compression ratio for kernels that utilize the hardware well. This is an expected result, as unused entities are exploited.

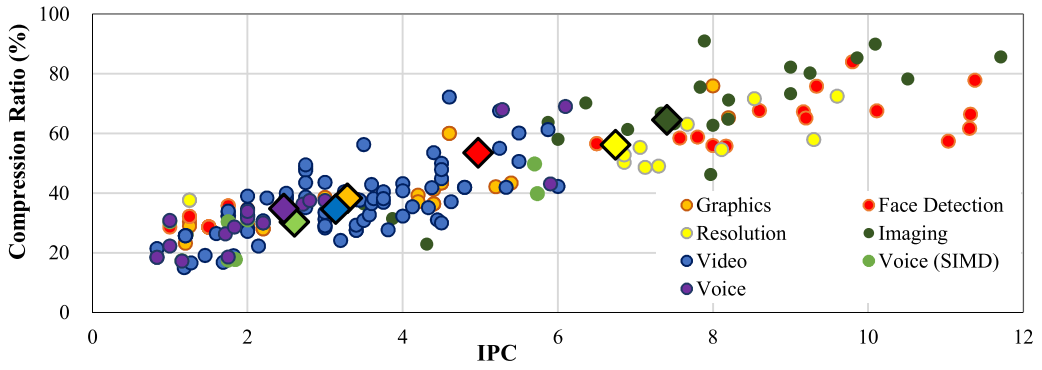

Fig. 19. Correlation between IPC and compression ratio for the different application domains.

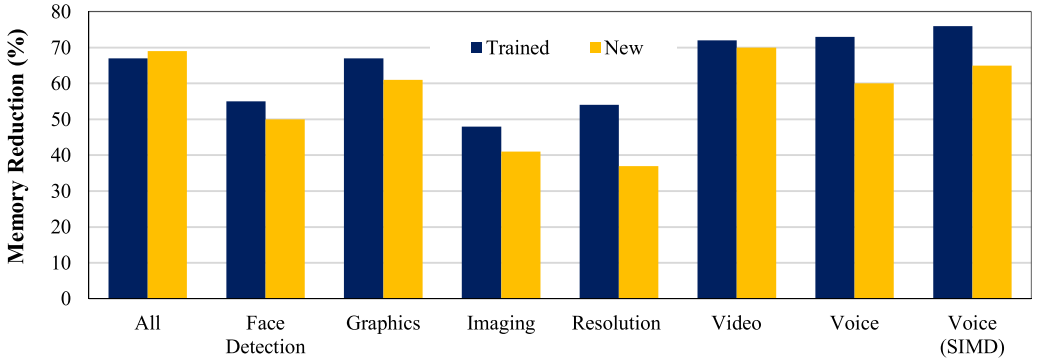

Fig. 20. Memory reduction per application domain for trained and new (untrained) applications.

It is, however, important to note that the applications used for this evaluation are used in production and have already been optimized for the SRP. Achieving higher utilization, such as through better compilation techniques, is not easy, because the limiting factor for most loops are loop-carried data dependencies that make it impossible to generate a loop encoding with fewer configuration lines.

7.6 Compressibility of New Code

A big advantage of CGRAs is their reconfigurability after shipping. Typically, the basic set of applications to be run on the CGRA is known at design time; however, updates to, for example, multimedia codecs or bugfixes may require downloading and execution of new code. We are thus interested in the compressibility and energy consumption of unseen code for the different application domains.

For the test, an 80:20 ratio of trained versus new code is used—that is, 80% of the code is assumed to be known and used by the statistical analyzer to compute a partitioning. The remaining 20% are then compiled with the fixed partitioning. Figure 20 shows the results for the different application domains. The result is the average of a fivefold cross validation—that is, the loops are divided into five parts of similar size in terms of the number of configuration lines. Each of the five parts serves once as the new code, whereas the other four parts make up the training set. For five of the eight tested domains, the compressibility of new code is reduced by less than 7%. Both versions of `Voice`

Table 3. Memory Savings Obtained by Cross Referencing the Partitioning for a Specific Domain With All Other Application Domains

	Test						
Train	Face Detection	Graphics	Imaging	Resolution	Video	Voice	Voice (SIMD)
Face Detection	**55**	+1	−32	−36	+4	+4	+2
Graphics	−37	**67**	−45	−44	−4	−7	−12
Imaging	−9	+6	**49**	−13	+12	+12	+11
Resolution	−23	−4	−19	**52**	+2	−3	−3
Video	−37	−11	−50	−51	**73**	−11	−14
Voice	−39	−17	−45	−47	−10	**72**	+5
Voice (SIMD)	−45	−25	−49	−49	−19	−12	**76**

Table 4. Memory Reduction of Bin Packing–Based Partitioning Versus Exhaustive Search

Partitions	**PEs**	**Combinations**	**Bin Packing (%)**	**Exhaustive Search (%)**	**Difference (%)**
8	6	262,144	70.43	70.43	0
6	8	1,679,616	68.96	69.08	0.12
4	10	1,048,576	62.07	63.53	1.46

and `Resolution` show larger drops with 11% , 13% , and 17% lower compressibility yet still achieve a significant memory and runtime energy reduction.

7.7 Compressibility of Code From Other Application Domains

The previous result shows that new code from the same application domain shows a reduced but still significant compressibility. To find out whether a partitioning generated for a certain application domain also achieves a good compression ratio for other domains, the partitionings for a specific application domain is cross referenced with all other domains. The results are shown in Table 3. For an application domain `app dom`, the columns show either the absolute memory savings when the partitioning is applied to `app dom` itself or the difference in as a percentage when applied to the other domains. We observe that the compressibility of applications is domain dependent—that is, compared to the trained new results from Figure 20, cross-domain compressibility is significantly worse. In other words, the static partitioning of the presented scheme is a limitation when applied to new applications with different code characteristics.

7.8 Optimality of Bin Packing–Based Partitioning

An important question is how well the bin packing–based algorithm performs compared to the optimal solution. We have performed an analysis of three different configurations with a limited number of processing units since an exhaustive search over all 16^{383} combinations for the 383 entities and 16 bins is impossible to compute. Table 4 displays the results for 8/6/4 partitions with 6/8/10 processing elements taken from `All loops` for both the bin packing–based partitioning and exhaustive search. Bin packing has a random component (the order in which the entities are assigned to the partitions), and hence the results of bin packing are the average of three distinct runs. The results show that, although not optimal, the presented greedy bin packing algorithm achieves results close to what is theoretically possible.

8 CONCLUSION

In this article, we presented a method to significantly reduce the energy consumption of the instruction fetch logic in CGRAs. The method exploits unused configuration signals in the encoding of the modulo schedule of a loop kernel to create consecutive duplicated configuration lines that are then eliminated by a compression scheme. To improve compressibility of code, a spatial optimization partitions the configuration memory in several partitions. A temporal optimization applied by the compiler minimizes signal changes before compressing the code. Decompression of the compressed configuration at runtime does not introduce additional latency. The decompression logic is able to significantly lower the dynamic energy consumption of the instruction fetch logic by reducing the number of reads from the configuration memory. The presented technique has been implemented in a production-level modulo scheduler for the SRP and evaluated on an existing 4x4 architecture with a wide range of loop kernels from real-world applications. The method achieves, on average, an energy reduction of 70% and a memory reduction of 63% for different application domains. The presented method is currently being applied to the production version of the SRP.

REFERENCES

[1] Nazish Aslam, Mark Milward, Ahmet Teyfik Erdogan, and Tughrul Arslan. 2008. Code compression and decompression for coarse-grain reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 12, 1596–1608. DOI: http://dx.doi.org/10.1109/TVLSI.2008.2001562

[2] Nazish Aslam, Mark Milward, Ioannis Nousias, Tughrul Arslan, and Ahmet Erdogan. 2007. Code compression and decompression for instruction cell based reconfigurable systems. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*. 1–7. DOI: http://dx.doi.org/10.1109/IPDPS.2007.370392

[3] Nazish Aslam, Mark Milward, Ioannis Nousias, Tughrul Arslan, and Ahmet Erdogan. 2007. Code compressor and decompressor for ultra large instruction width coarse-grain reconfigurable systems. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*. 297–298. DOI: http://dx.doi.org/10.1109/FCCM.2007.28

[4] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. 2003. PACT XPP—a self-reconfigurable data processing architecture. *Journal of Supercomputing* 26, 2, 167–184. DOI: http://dx.doi.org/10.1023/A:1024499601571

[5] Bruno Bougard, Bjorn De Sutter, Diederik Verkest, Liesbet Van der Perre, and Rudy Lauwereins. 2008. A coarse-grained array accelerator for software-defined radio baseband processing. *IEEE Micro* 28, 4, 41–50. DOI: http://dx.doi.org/10.1109/MM.2008.49

[6] Kenneth M. Butler, Jayashree Saxena, Atul Jain, Tony Fryars, Jack Lewis, and Graham Hetherington. 2004. Minimizing power consumption in scan testing: Pattern generation and DFT techniques. In *Proceedings of the 2004 International Conference on Test*. 355–364. DOI: http://dx.doi.org/10.1109/TEST.2004.1386971

[7] Moo-Kyoung Chung, Yeon-Gon Cho, and Soojung Ryu. 2012. Efficient code compression for coarse grained reconfigurable architectures. In *Proceedings of the IEEE 30th International Conference on Computer Design (ICCD'12)*. IEEE, Los Alamitos, CA, 488–489.

[8] Moo-Kyoung Chung, Jun-Kyoung Kim, Yeon-Gon Cho, and Soojung Ryu. 2013. Adaptive compression for instruction code of coarse grained reconfigurable architectures. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'13)*. IEEE, Los Alamitos, CA, 394–397.

[9] Thomas M. Conte, Sanjeev Banerjia, Sergei Y. Larin, Kishore N. Menezes, and Sumedh W. Sathaye. 1996. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*. 201–211. DOI: http://dx.doi.org/10.1109/MICRO.1996.566462

[10] Bernhard Egger, Hochan Lee, Duseok Kang, Mansureh S. Moghaddam, Youngchul Cho, Yeonbok Lee, Sukjin Kim, Soonhoi Ha, and Kiyoung Choi. 2017. A space- and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO'17)*. IEEE, Los Alamitos, CA, 197–209. http://dl.acm.org/citation.cfm?id=3049832.3049854.

[11] Nasim Farahini, Ahmed Hemani, Hassan Sohofi, Syed M. A. H. Jafri, Muhammad Adeel Tajammul, and Kolin Paul. 2014. Parallel distributed scalable runtime address generation scheme for a coarse grain reconfigurable computation and storage fabric. *Microprocessors and Microsystems* 38, 8, 788–802. DOI: http://dx.doi.org/10.1016/j.micpro.2014.05.009

[12] Bita Gorjiara and Daniel Gajski. 2007. FPGA-friendly code compression for horizontal microcoded custom IPs. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA'07)*. ACM, New York, NY, 108–115. DOI: http://dx.doi.org/10.1145/1216919.1216935

[13] Paul M. Heysters, Gerardus J. M. Smit, and Egbert Molenkamp. 2003. Montium—balancing between energy-efficiency, flexibility and performance. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03)*. 235–241.

[14] Nagisa Ishiura and Masayuki Yamaguchi. 1997. Instruction code compression for application specific VLIW processors based on automatic field partitioning. In *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies*. 105–109.

[15] Syed M. A. H. Jafri, Ahmed Hemani, Kolin Paul, Juha Plosila, and Hannu Tenhunen. 2011. Compression based efficient and agile configuration mechanism for coarse grained reconfigurable architectures. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. 290–293. DOI: http://dx.doi.org/10.1109/IPDPS.2011.166

[16] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. 2002. The imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 282–288. DOI: http://dx.doi.org/10.1109/ICCD.2002.1106783

[17] Sami Khawam, Ioannis Nousias, Mark Milward, Ying Yi, Mark Muir, and Tughrul Arslan. 2008. The reconfigurable instruction cell array. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 1, 75–85. DOI: http://dx.doi.org/10.1109/TVLSI.2007.912133

[18] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. 2012. ULP-SRP: Ultra low power Samsung reconfigurable processor for biomedical applications. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'12)*. IEEE, Los Alamitos, CA, 329–334. DOI: http://dx.doi.org/10.1109/FPT.2012.6412157

[19] Yoonjin Kim, Rabi N. Mahapatra, Ilhyun Park, and Kiyoung Choi. 2009. Low power reconfiguration technique for coarse-grained reconfigurable architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 5, 593–603.

[20] Yoonjin Kim, Ilhyun Park, Kiyoung Choi, and Yunheung Paek. 2006. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'06)*. ACM, New York, NY, 310–315. DOI: http://dx.doi.org/10.1145/1165573.1165646

[21] Monica Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI'88)*. ACM, New York, NY, 318–328. DOI: http://dx.doi.org/10.1145/53990.54022

[22] A. Lambrechts, P. Raghavan, M. Jayapala, F. Catthoor, and D. Verkest. 2005. Energy-aware interconnect-exploration of coarse grained reconfigurable processors. In *Proceedings of the Workshop on Application Specific Processors*.

[23] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoung Choi. 2009. FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability. In *Proceedings of the 2009 International Conference on Field-Programmable Technology*. 376–379. DOI: http://dx.doi.org/10.1109/FPT.2009.5377609

[24] Jaedon Lee, Youngsam Shin, Won-Jong Lee, Soojung Ryu, and Kim Jeongwook. 2013. Real-time ray tracing on coarse-grained reconfigurable processor. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'13)*. 192–197. DOI: http://dx.doi.org/10.1109/FPT.2013.6718352

[25] Won-Jong Lee, Shi-Hwa Lee, Jae-Ho Nah, Jin-Woo Kim, Youngsam Shin, Jaedon Lee, and Seok-Yoon Jung. 2012. SGRT: A scalable mobile GPU architecture based on ray tracing. In *ACM SIGGRAPH Talks*. ACM, New York, NY, 44.

[26] Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference*. ACM, New York, NY, 109–119.

[27] Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Hyun-Sang Park, Seokyoon Jung, and Shihwa Lee. 2013. A novel mobile GPU architecture based on ray tracing. In *Proceedings of the 2013 IEEE International Conference on Consumer Electronics (ICCE'13)*. IEEE, Los Alamitos, CA, 21–22.

[28] Haris Lekatsas, Jörg Henkel, and Venkata Jakkula. 2002. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. ACM, New York, NY, 34–39. DOI: http://dx.doi.org/10.1145/513918.513929

[29] Shuo Li, Nasim Farahini, Ahmed Hemani, Kathrin Rosvall, and Ingo Sander. 2013. System level synthesis of hardware for DSP applications using pre-characterized function implementations. In *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*. IEEE, Los Alamitos, CA, Article 16, 10 pages. http://dl.acm.org/citation.cfm?id=2555692.2555708

[30] Zhiyuan Li and Scott Hauck. 1999. Don't care discovery for FPGA configuration compression. In *Proceedings of the 1999 ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA'99)*. ACM, New York, NY, 91–98. DOI: http://dx.doi.org/10.1145/296399.296435

[31] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*. 61–70. DOI: http://dx.doi.org/10.1007/978-3-540-45234-8_7

[32] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings—Computers and Digital Techniques* 150, 5, 255–261. DOI: http://dx.doi.org/10.1049/ip-cdt:20030833

[33] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report HPL-2009-85. HP Laboratories, 22–31.

[34] Chetan Murthy and Prabhat Mishra. 2009. Bitmask-based control word compression for NISC architectures. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI (GLSVLSI'09)*. ACM, New York, NY, 321–326. DOI: http://dx.doi.org/10.1145/1531542.1531616

[35] T. Nishimura, K. Hirai, Y. Saito, T. Nakamura, Y. Hasegawa, S. Tsutsusmi, V. Tunbunheng, and H. Amano. 2008. Power reduction techniques for dynamically reconfigurable processor arrays. In *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications*. 305–310. DOI: http://dx.doi.org/10.1109/FPL.2008.4629949

[36] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. 2009. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*. ACM, New York, NY, 21–30. DOI: http://dx.doi.org/10.1145/1542452.1542456

[37] Seongsik Park and Kiyoung Choi. 2011. An approach to code compression for CGRA. In *Proceedings of the 2011 3rd Asia Symposium on Quality Electronic Design (ASQED'11)*. IEEE, Los Alamitos, CA, 240–245.

[38] Yongjun Park, Hyunchul Park, and Scott Mahlke. 2009. CGRA express: Accelerating execution using dynamic operation fusion. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. ACM, New York, NY, 271–280. DOI: http://dx.doi.org/10.1145/1629395.1629433

[39] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 389–402. DOI: http://dx.doi.org/10.1145/3079856.3080256

[40] Marc Quax, Jos Huisken, and Jef van Meerbergen. 2004. A scalable implementation of a reconfigurable WCDMA rake receiver. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'04)*. IEEE, Los Alamitos, CA, 30230–. http://dl.acm.org/citation.cfm?id=968880.969243.

[41] B. Ramakrishna Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*. ACM, New York, NY, 63–74. DOI: http://dx.doi.org/10.1145/192724.192731

[42] Samsung. 2011. Samsung Exynos 4210. Retrieved February 13, 2018, from http://www.samsung.com/us/business/oem-solutions/pdfs/Exynos_v11.pdf.

[43] Muhammad Ali Shami and Ahmed Hemani. 2010. Control scheme for a CGRA. In *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing*. 17–24. DOI: http://dx.doi.org/10.1109/SBAC-PAD.2010.12

[44] Youngsam Shin, Jaedon Lee, Won-Jong Lee, Soojung Ryu, and Jeongwook Kim. 2014. Full-stream architecture for ray tracing with efficient data transmission. In *Proceedings of the 2014 IEEE International Symposium on Circuits and Systems (ISCAS'14)*. IEEE, Los Alamitos, CA.

[45] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M. Chaves Filho. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers* 49, 5, 465–481.

[46] Dongkwan Suh, Kiseok Kwon, Sukjin Kim, Soojung Ryu, and Jeongwook Kim. 2012. Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'12)*. 67–70. DOI: http://dx.doi.org/10.1109/FPT.2012.6412114

[47] Synopsys. 2010. Synopsys Design Compiler 2010. Available at http://www.synopsys.com/.

[48] Panagiotis Theocharis and Bjorn De Sutter. 2016. A bimodal scheduler for coarse-grained reconfigurable arrays. *ACM Transactions on Architecture and Code Optimization* 13, 2, Article 15, 26 pages. DOI: http://dx.doi.org/10.1145/2893475

[49] Vasutan Tunbunheng, Masayasu Suzuki, and Hideharu Amano. 2005. RoMultiC: Fast and simple configuration data multicasting scheme for coarse grain reconfigurable devices. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*. IEEE, Los Alamitos, CA, 129–136.

[50] Andrew Wolfe and Alex Chanin. 1992. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO -25)*. IEEE, Los Alamitos, CA, 81–91. http://dl.acm.org/citation.cfm?id=144953.145003.

Received June 2017; revised September 2017; accepted November 2017