A Space- and Energy-Efficient Code Compression/Decompression Technique for Coarse-Grained Reconfigurable Architectures

Bernhard Egger[1], Hochan Lee[1], Duseok Kang[1], Mansureh S. Moghaddam[1], Youngchul Cho[2], Yeonbok Lee[2], Sukjin Kim[2], Soonhoi Ha[1], Kiyoung Choi[1]

[1]Seoul National University, Seoul, Korea
[2]Samsung Electronics, Seoul, Korea
[1]{bernhard,nicelhc13,kangds0829,mansureh,sha,kchoi}@snu.ac.kr
[2]{rams.cho,yeonbok.lee,sukj.kim}@samsung.com

Abstract

We present an effective code compression technique to reduce the area and energy overhead of the configuration memory for coarse-grained reconfigurable architectures (CGRA). Based on a statistical analysis of existing code, the proposed method reorders the storage locations of the reconfigurable entities and splits the wide configuration memory into a number of partitions. Code compression is achieved by removing consecutive duplicated lines in each partition. Compressibility is increased by an optimization phase in the compiler. The optimization minimizes the number of configuration changes for individual reconfigurable entities. Decompression is performed by a simple hardware decoder logic that is able to decode lines with no additional latency and negligible area overhead. Experiments with over 190 loop kernels from different application domains show that the proposed method achieves a memory reduction of over 40% on average with four partitions. The compressibility of yet unseen code is only slightly lower with 35% on average. In addition, executing compressed code results in a 22 to 47% reduction in the configuration logic's energy consumption.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Other Architecture Styles; D.3.4 [*Programming Languages*]: Compilers; G.1.6 [*Numerical Analysis*]: Optimization

Keywords Coarse-grained reconfigurable architecture, code compression, energy reduction

1. Introduction

The trend of high-resolution displays and high-quality audio on mobile devices has lead 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). While the latter provide high performance at a relatively low power consumption, the prohibitively high development cost and the limited programmability hinders broad applicability. FPGAs, on the other hand, offer very high flexibility through bit-level reconfigurability at the expense of a high routing overhead and a long reconfiguration phase. Coarse-grained reconfigurable architectures (CGRA) fill this gap by providing the necessary computational power at a moderate energy-consumption while remaining easily programmable [16, 21]. CGRAs have been integrated in a number of commercially-available mobile systems such as smartphones, tablets, or smart TVs [5, 11, 12, 20].

CGRA processors are built around an array of heterogeneous processing elements. Data memory, register files and constant units hold and produce values. An interconnection network, composed of multiplexers, latches and wires, routes data from producers to consumers. CGRAs typically execute loop kernels and operate in data-flow mode, i.e., no control flow operations are supported. The data-flow graph (DFG) of a loop kernel is mapped onto the CGRA using software pipelining. Modulo scheduling or simulated annealing are two widely used variants to compute these temporal mappings [17, 19]. The generated instruction streams are stored in the CGRA's configuration memory, a wide on-chip SRAM. Each line of the configuration memory holds the configuration signals for every reconfigurable hardware entity for a given execution cycle. The large number of reconfigurable entities requires memories that are often more than 1,000 bits wide and several hundred lines deep, thus occupying a significant amount of the entire chip area. In addition, the frequent reads from the configuration memory pose a burden on energy consumption as well, amounting to 25-45% of the entire CGRA chip energy consumption [6, 10].

In this paper, we introduce an efficient code compression technique for high-performance CGRAs. We apply two techniques that render the code suitable for temporal deduplication. The first technique is an optimization in the code generator of a modulo scheduler that minimizes the number of configuration changes of the hardware entities. The second technique involves reorganizing the wide config-

uration memory into several sub-partitions, each holding distinct subsets of the configuration. Based on a statistical analysis of code, the configurations of hardware entities that frequently change in the same cycles are allocated to the same partition which, in turn, increases the compressibility of data stored in to partitions. The reorganized code exhibits very good compression rates of 70% on average when applied to single loops. A simple hardware decoder reassembles the configuration lines from the individual partitions at runtime. The decoder logic is extremely lightweight and does not introduce any additional latency.

We have applied the proposed method to the compiler and architecture of a commercially available CGRA, the Samsung Reconfigurable Processor [27] deployed in smartphones, TVs, printers, and cameras of the same manufacturer [13, 14, 23, 25]. We employ a heuristic, a genetic algorithm and an integer linear programming (ILP) formulation to generate the partitioning and observe very good code compressibility and reduction in energy consumption for known and also new loop kernels.

In summary, this paper makes the following contributions.

- we propose a compiler optimization that generates compression-friendly code based on a statistical analysis.
- we design a partitioned configuration memory that allows decompression at real-time with no additional latency.
- we implemented the compiler optimization into the compiler framework and applied the partitioned configuration memory to the hardware of a commercial CGRA chip.
- we show a reduction of 28 to 56% in code size and 22 to 47% in energy consumption on average for known and new loop kernels of real-world applications.

The remainder of this paper is organized as follows. Section 2 discusses related work. The organization of CGRAs and code generation therefor is briefly discussed in Section 3. Section 4 gives an overview of the proposed technique and discusses the temporal and spatial code optimization techniques. Section 5 describes the partitioning algorithms. Section 6 compares the different partitioning approaches and discusses the results. Section 7, finally, concludes this paper.

2. Related Work

There exists a significant amount of work aiming at reducing the storage and energy requirements of caches and configuration memories of long-instruction-word processors such as VLIW and CGRA processors. The techniques can be broadly divided into two categories: techniques that eliminate duplication in the instruction stream and methods that compress the instruction stream using dictionary-based compression schemes. Compression rate and memory reduction are interchangeably used metrics. The compression rate is computed by dividing the size of the compressed code by the original code size, i.e., lower is better. The memory reduction is the complement and reports by how much the storage requirements can be reduced. Memory reduction is obtained by subtracting the compression rate from 1, i.e., higher is better.

The first body of work eliminates duplication in the instruction stream by allowing entire configuration lines or individual hardware units to keep their configuration over several cycles. These methods are typically applied to CGRAs with a large number of processing units (PEs) and use simple code mapping schemes. MorphoSys [26] is one of the earlier CGRA designs and focuses on regular, data-parallel code. Configuration words are broadcast to an entire row or column of PEs that then all execute the same operation. Rows and columns can be configured separately and individually to reduce the amount of reconfiguration needed. RoMultiC [30] relaxes these limitations by allowing individual PEs to be configured. Configuration data is broadcast through the interconnection network, and a separate multicast bitmap defines which PEs are to be reconfigured. The selective reconfigurability allows the authors to eliminate duplicated pieces of code. The long overhead of reconfiguration (tens of cycles) limits the applicability of this approach to our high-speed CGRA. A more flexible approach is presented in [6, 7]. Code is mapped either spatially, i.e., no reconfigurations are required through the entire lifetime of a loop, or temporally. Spatial mappings are very energy efficient, however, only exist but for the very simplest of loops. For a more flexible temporal mapping in which the columns from left to right execute the same operations with a one-cycle delay, the authors propose to send the configuration data only to the first column on the left and then pass on the configuration word to the next column in every cycle. Savings of up to 87% are reported. A similar approach is presented in [15], here data and instructions flow from row-to-row. For specialized codes the authors report memory savings of around 84%. For CGRAs using more powerful modulo scheduling techniques these techniques are not applicable due to the highly irregular nature of modulo scheduled loop code.

Code compression based on dictionary-based compression schemes have also been proposed. A Lempel-Ziv-based approach is presented in [8] and achieves memory savings of up to 50%, however, the long decoder overhead, limiting the clock frequency to 200MHz, renders this approach unsuitable for high-speed CGRAs. A dictionary-based code compression schemes for a 4x4 CGRA has been proposed in [2, 3]. Similar to our work, temporal duplication of configuration signals for individual entities are detected. In addition, the scheme also compresses along the spatial axis: if two neighboring PEs execute the same operation, the configuration is only stored once. Reported compression rates are around 60% in experiments with one single application.

The work most closely related to ours is presented in [1]. Based on a dictionary-based compression scheme, the au-

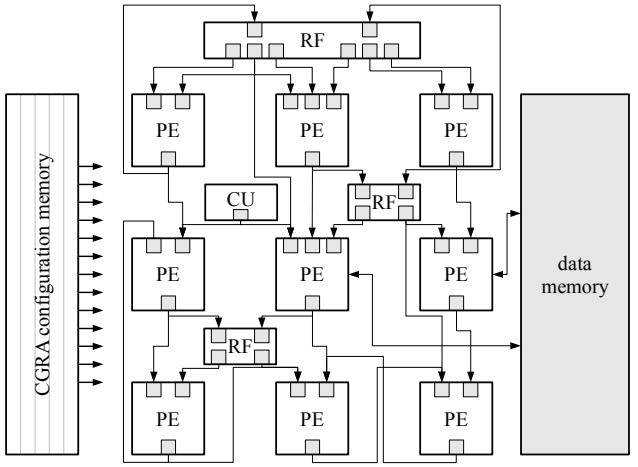


Figure 1. A coarse-grained reconfigurable architecture.

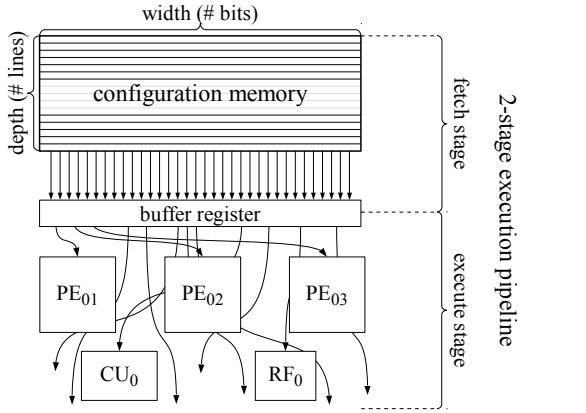


Figure 2. Configuration memory organization.

thors propose to reorganize the spatial organization of the entities in a configuration line to render the code more compressible. Good compression rates of as low as 40% are reported for flexible code mapping schemes. The scheme only tackles compression of the configuration of the 64 PEs and cannot easily be applied to register files, constant units and the interconnection network configuration. This limits the applicability of the approach to our target architecture since the configuration of the PEs only comprises 15% of the entire configuration line.

We propose a technique that is able to compress and decompress complex code patterns for CGRAs that allow individual configuration of PEs and other reconfigurable hardware entities in every cycle. The small number of PEs of our 4x4 target architecture and the high density of non-`nop` operations renders compression more challenging. Our compression technique does not introduce any additional latency into the decompression cycle, rendering it an ideal candidate for high-performance CGRAs.

3. Background

The following paragraphs provide the necessary background of CGRA architectures and the code generation process.

3.1 Architecture

The computational power of CGRAs is provided by a number of **processing elements** (PE) capable of executing word-level operations on scalar, vector, or floating point data. PEs are often heterogeneous in their architecture and functionality. A number of **register files** (RF) provide temporary and fast storage of data values. Unlike traditional ISAs, immediate operand values are not encoded directly in an instruction. Instead, **constant units** (CU) are used to generate constant values whenever needed. Figure 1 shows an example of a fictional CGRA with nine PEs, three register files, and one constant unit.

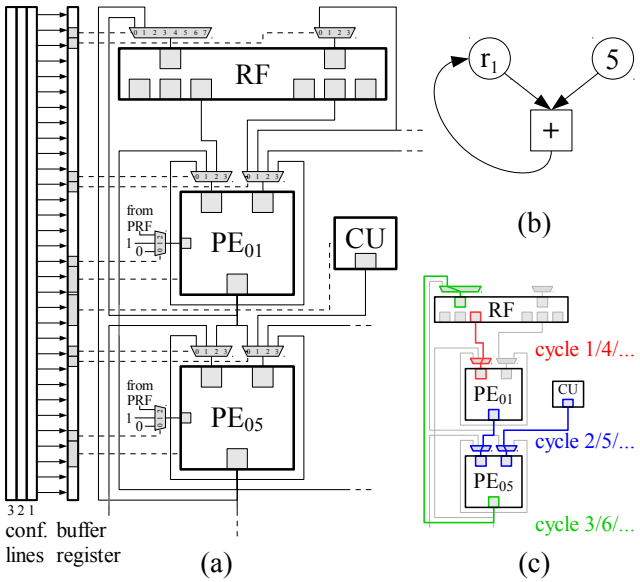
Input operands and results of PEs are routed through an **interconnection network** comprising physical connections (wires), multiplexers, and latches. The interconnection network is typically sparse and irregular. Separate networks exist to carry the different data types through the CGRA.

PEs support **predicated execution**, i.e., depending on a one-bit input signal, the PE will either execute the operation or perform a no-operation (`nop`). PEs can also generate predicate signals that then control execution of operations on other PEs. For this purpose, predicate register files and a separate predicate interconnection network exists in CGRAs.

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 RFs' write enable and read port signals, the immediate values for CUs, and the selection signal for each of the multiplexers in the interconnection network. Configuration line widths of several hundred bits are the norm - a configuration line of the Samsung SRP with 4x4 PEs, for example, is over 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 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.

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



config. line	cycle	PE01 op	PE01 in0	PE01 in1	PE05 op	PE05 in0	PE05 in1	RF adr	RF wp0 we	RF sel	RF rp2 adr	CU val
1	1	mov	2						0		r1	
2	2				add	2	1		0			5
3	3							r1	1	1		
1	4	mov	2						0		r1	
2	5				add	2	1		0			5
...	...											

(d)

Figure 3. Encoding a DFG in the configuration lines.

3.3 Execution Model

CGRAs execute software-pipelineable loops [9]. The kernel of a loop is encoded as a number of 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 by, for example, 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. Compilers for CGRAs employ modulo scheduling [22], a form of software-pipelining to map code onto a CGRA.

For input operands the compiler computes and selects a route through the interconnection network by setting the selection bits for the relevant multiplexers. As an example, consider the closeup of another fictional 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 data flow graph is shown in Figure 3 (b). The constant 5 can only be produced by the constant unit 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 1 cycle. In cycle 2, PE_{05} selects the output of PE_{01} as operand 1 ($= r_1$) and the output of the CU ($= 5$) on the second input operand, then adds the two. The output is produced one cycle later, at time 3 and written back to the register file, then execution begins anew. The compiler generates all the 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 sub-components `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 constant unit `CU`. Empty cells denote entities that are not active in the respective cycle.

3.4 Area and Energy Breakdown

In typical CGRA designs, the configuration memory accounts for 10-20% of the chip area and consumes 25-45% of the total chip energy consumption [6, 10]. The comparatively high energy consumption is due to the fact that a new configuration line is read from the configuration memory for every execution cycle. The presented technique is able to reduce the area overhead by 40 and the energy consumption by 35 to 45 percent; the following sections discuss how.

4. Compression/Decompression Technique

Requiring decompression at clock frequencies of 1 GHz and above precludes compression schemes that require complex re-composition of configuration lines at runtime. In particular, this requirement rules out the popular dictionary-based schemes because decompression of a single line requires serial decompression of the compressed bit patterns.

4.1 Compression and Decompression

At the heart of the proposed configuration memory optimization lies a simple compression scheme that eliminates temporal redundancy in the form of duplicated consecutive lines. Figure 4 illustrates the idea. The uncompressed code contains five configuration lines. Line 2 is a duplicate of line 1, lines 4 and 5 are identical to line 3. After compression only two lines remain as shown in the lower right part of the figure.

Decompression is performed on the compressed configuration using a bit vector denoted *decompression offset*. The 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' in position 1 and 3, representing the original indices of the configuration lines. The program counter PC repeatedly iterates from

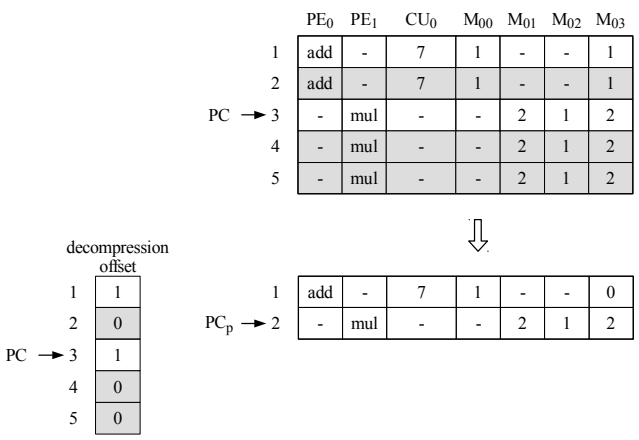


Figure 4. Deduplication of consecutive identical lines.

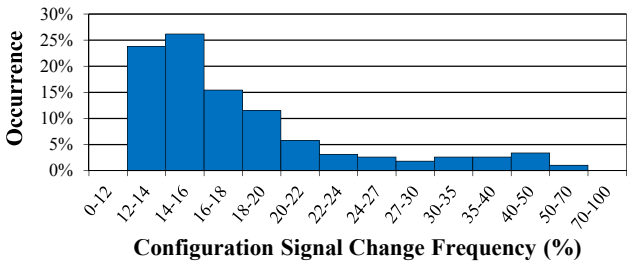


Figure 5. Histogram of signal changes per entity.

index 1 to 5 in the original code until the loop ends. Decompression requires an additional counter, denoted PC_p. The original program counter PC iterates through lines 1 to 5 in the decompression offset while PC_p points to the currently active line in the compressed table. For a given loop index t, $PC_p(t)$ is computed by adding the value of the decompression offset $dofs$ to $PC_p(t-1)$ as follows

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

Note that the loop index space starts at 1.

4.2 Improving Compressibility

The proposed scheme does not yield significant compression for typical loop kernel code and CGRA configuration memory architectures. Experiments with 193 loop kernels for a 4x4 variant of the Samsung Reconfigurable Processor revealed that in the total of over 1,400 configuration lines there exists not one 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. The probability that a single entity e performs the exact same function in two consecutive cycles is $p(e) \leq 1$. For an entire line l to remain constant with respect to its preceding line the probability is $p(l) = \prod_{e=1}^{N} p(e)$. Figure 5 shows the distribution of the signal change frequency for all 1,417 configuration lines. Over 75% of the entities are reconfigured with a probability of 12-20%, i.e., once every 5 to 10 lines. Assuming entities are independent random variables, the probability of getting a consecutive duplicated line is $3.3 \cdot 10^{-18}$. Of course this is overly pessimistic since the configuration signals of entities are not completely independent, but even with an average configuration change of 1 in 20, we can only expect three duplicates per one billion configuration lines.

The second reason why the simple compression scheme does not work is that modulo schedulers do not generate "compression-friendly" code. Take, for example, a multiplexer in the interconnection network that selects input 1 in cycle 1 and input 2 in cycle 3. No data is routed through the multiplexer in cycles 2, 4, and 5, so the the modulo scheduler issues no selection signal for these cycles. The code generator, having to encode a selection signal for every cycle of the loop, outputs the signal 0 for inactive cycles, yielding the signals 1, 0, 2, 0, 0 for the multiplexer configuration of the loop's five cycles. This particular encoding allows elimination of only one line (line 5 is identical to line 4).

Based on these observations, we propose and implement two techniques, a *temporal* and a *spatial* optimization, that render configuration code for CGRAs more compressible. 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 where the entity is idle. The spatial optimization is based on 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 the techniques in more detail.

4.3 Temporal Optimization

The temporal optimization exploits two properties of CGRAs. 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 files ports are not used and remain unconfigured. Second, the data values generated by inactive entities have no impact on the computation since the compiler only routes data from producers to consumers of the source code's dataflow graph. As a consequence, configuration signals of inactive entities can be set to values that improve temporal duplication.

As a motivating example, consider a configuration line comprising only a single PE that is activated in cycles 2 and 4 of a 5-cycle loop executing a `mov` and `add` operation, respectively. The standard way to encode this configuration sequence for the five cycles is `nop`, `mov`, `nop`, `add`, and `nop`. There are no consecutive duplicated lines, so compression

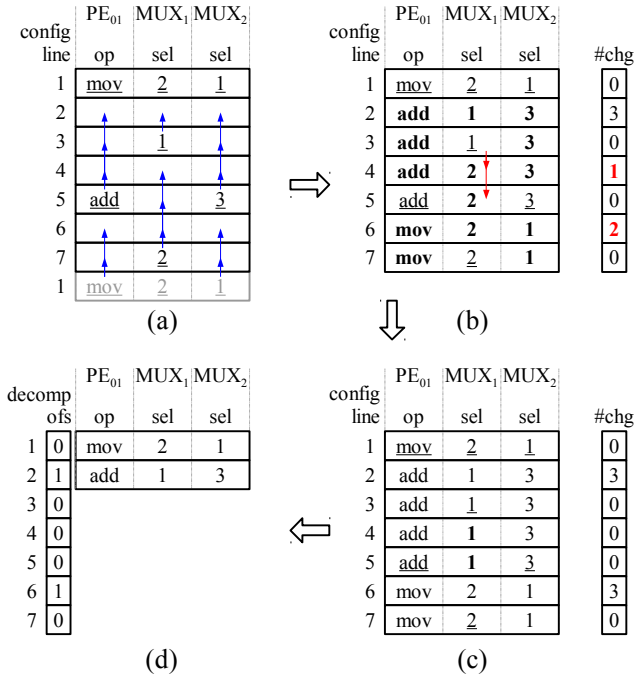


Figure 6. The ASAP-ALAN propagation algorithm.

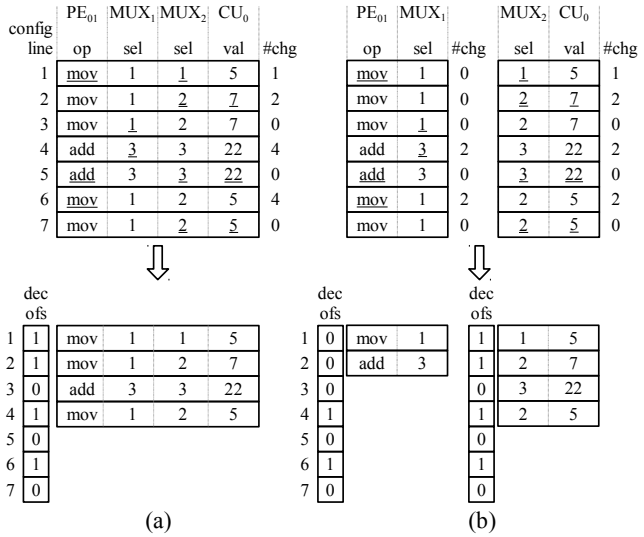


Figure 7. Improving compressibility by partitioning.

fails. We can minimize the number of configuration changes by replacing the `nop` operation with `mov` or `add` operations. One possible encoding is `mov`, `mov`, `mov`, `add`, `add`. The code still performs the same computation (results generated in cycles 1, 3, and 5 are never used), but compression is now able to remove configuration lines 2 and 3 (identical to 1) and 5 (duplicate of 4), yielding a memory reduction of 60%.

Configuration lines, even when partitioned, comprise the configuration signals of several entities, not just one. To maximize compressibility, the signal changes of the individual entities should occur in the same line(s). We employ a two-step algorithm that starts with a standard modulo schedule 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 upwards (backwards 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 1 continues at cycle m. Figure 6 (a)-(b) shows the original modulo schedule before and after ASAP propagation (blue arrows). If the number of configuration changes per line (denoted #chg) with respect to the preceding one is 0, the entire line can be eliminated (lines 1, 3, 5, and 7).

Maximal compression is achieved if the number of changes per line is either 0 (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 4 with 1 and line 6 with 2 changes. If the entity causing the change remains constant from the first to the second line and is inactive in all cycles, we can propagate the configuration of the line preceding the first suboptimal line down to the second one. This step is called ALAN propagation (as-late-as-necessary, red arrows). Consider $\texttt{MUX}_1$. It causes the only signal change in line 4, but remains constant until and including the next line with > 0 configuration changes, line 6. The ALAN step propagates the signal of line 3 down to line 5, causing $\texttt{MUX}_1$ to switch in line 6. This transformation is shown in Figure 6 (c). The result is optimal as the number of changes is either 0 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 `nop` operations of PEs with ALU operations increases the dynamic energy consumption. We prevent execution of idle non-`nop` `operations` by setting the predicated execution bit of the PE to 0, effectively blocking execution if the operation.

4.4 Spatial Optimization

The spatial optimization improves compressibility by splitting the configuration line into several *partitions*. Each partition can then be compressed separately. Figure 7 shows an example. The code after temporal optimization is shown in the upper part, the corresponding compressed configuration along with the decompression offset in the lower part of Figure 7 (a). Assuming each signal of the four entities occupies 4 bits, the size of the configuration memory is reduced from $7 * 4 * 4 = 112$ bits to $4 * 4 * 4 = 64$ bit. Partitioning the 4-entity wide line into two partitions containing two entities each (Fig. 7 (b)), the first partition can be compressed with only two lines whereas the second partitions still requires four. Thanks to this separation, the to-

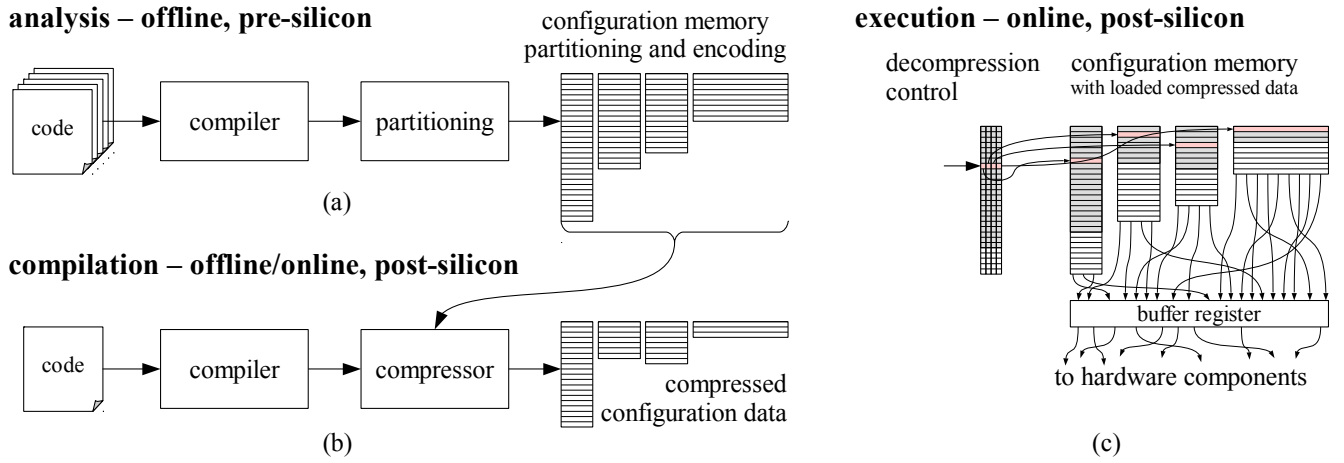


Figure 8. Analysis, compilation, and execution.

tal size of the compressed configuration shrinks from 64 to $2 * 2 * 4 + 4 * 2 * 4 = 48$ bit.

It is important to note that spatial optimization is static, i.e., the partitions are determined at design time of the chip and cannot be modified thereafter. In addition, partitioning introduces a minimal overhead. Since each partition can be compressed individually, a separate decompression offset table needs to be generated for each partition. At runtime, additional program counters are required to enable independent reads from the different partitions.

5. Partitioning Based on Statistical Analysis

The compression technique described in the previous section achieves very high compression rates when applied to individual loops. The average compression ratio for the 193 loop kernels is less than 0.3, i.e., over 70% of memory savings can be achieved. In reality, however, CGRA chips execute a number of application each comprising one or several loop kernels. Since the proposed 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 partitioning scheme and encode each hardware entity into its pre-determined partition.

Figure 8 illustrates the three distinct phases of the proposed technique. In the **analysis** phase, loop kernels are compiled using a modulo scheduler generating temporally-optimized code. A partitioning algorithm then computes a partitioning that maximizes compressibility for the given loops. We have implemented and describe three techniques, a scheme based on heuristics, one based on a Genetic Algorithm, and an integer linear programming (ILP) formulation. The analysis and partitioning is performed *pre-silicon* as part of the optimization process of a CGRA to a certain application domain.

The compiler takes a given partitioning as an extra input during **compilation**. First, the configuration signals are encoded in the pre-determined partitions, then temporal optimization is applied. The individual partitions are then compressed by removing consecutive duplicated lines, and the decompression offsets are generated for each partition.

At runtime, the compressed code and the decompression offsets are loaded from the application binary into the configuration memory before **execution** of the loop kernel(s).

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 propose a simple, but very fast heuristic based on string edit distances and a more powerful but computationally more demanding genetic algorithm. Finally, we compare the results of both algorithms to those of an ILP formulation and show that both the heuristic and the GA achieve results close to the optimal result.

The input to the algorithms is the desired number of partitions n and the configuration lines of a (set of) loop(s), optimized as described in Section 4.3. The output is a partitioning of the configuration line into 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.

5.1.1 Heuristics Based on Edit Distance

For each individual entity $e \in E$ the complete sequence of configuration signals is first converted to a bit-vector denoted *change vector*. A 1 at position t denotes that the configuration value has changed from the previous cycle $t-1$ to the current one; a 0 implies that the configuration has remained the same.

The change vectors are then sorted by an augmented edit distance between the unified change vector of all already sorted entities and the change vector to be added. The augmented edit distance 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. The first element is set to the vector with the least number of $1s$, then one vector after the other is added. This sorting guarantees an order that is (a) incremental in the number of 1 bits in the change vectors and (b) places entities first that cause the slowest increase of configuration lines needed to encode all placed entities.

In a third step, the potential cut-off positions that divide the configuration line into partitions are identified. The sorted list of change vectors is scanned from the first to the last position. In each step the possible memory savings are computed by multiplying the bitwidth of the included entities by the number of $0s$ 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 potential separation point.

As the last step in the heuristics, up to $n-1$ cut-off positions yielding the highest memory savings are selected from the set of potential cut-off points. If there are less potential cut-off positions than requested partitions, the configuration line is split at random positions.

5.1.2 Genetic Algorithm

The second splitting algorithm is a genetic algorithm [4] implemented as follows. Genes are represented by integer arrays. Each item of the array corresponds to a configurable hardware entity of the CGRA, and its value defines the assigned partition. Preprocessing is performed by sorting the entities using the augmented editing distance causing resources with similar compression patterns to be located next to each other. The fitness of a solution is given by its compression ratio.

The next generation of genes is generated as follows. Selection, crossover, and replacement are performed as described in Sastry *et al.* [24]. To avoid getting stuck in local optima, the following mutations are performed:

1. Randomly select two partitions, p_1 and p_2, then move resources from p_1 to p_2 with probability 87.5%.
2. Randomly select a destination partition p, a length l, and a starting position pos. All entities from pos up to l are moved from the current partition to p.

We run 20 instances of the GA and select the best solution.

5.2 Integer Linear Programming Formulation

The problem is to find a division of the configuration stream into n vertical partitions such that the number of (re)configuration lines is minimized. For N reconfigurable entities and a schedule of M configuration lines, the inputs to the ILP formulation are a vector $W = \{w_j | 1 \leq j \leq N\}$ holding the bit width w_j of each reconfigurable entity j and the change matrix $V = \{v_{ij} | 1 \leq i \leq M, 1 \leq j \leq N\}$ indicating for every entity j and execution cycle i whether the configuration has changed with respect to the preceding cycle. The output are two matrices X and D where X holds the mappings of resources to partitions and the elements in D denote whether a configuration line in a partition can be eliminated or not. x_{jp} and d_{ip} are defined as follows.

$$x_{jp} = \begin{cases} 1, & \text{if resource } j \text{ is mapped to partition } p. \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

and

$$d_{ip} = \begin{cases} 1, & \text{if line } i \text{ in partition } p \text{ is all zero.} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

According to the above definition, the overall saving is equal to the weight of the lines containing only zero values for all partitions, represented as τ. The objective is to maximize τ.

$$\tau = \sum_p (\sum_i (\sum_j x_{jp} \cdot w_j) \cdot d_{ip}) \tag{3}$$

The set of constraints for the ILP problem is stated in inequalities 4 to 7. Based on the constraints 4 and 5, x_{jp} and d_{ip} are binary objective variables.

$$0 \leq x_{jp} \leq 1, \ \forall j, p \tag{4}$$

$$0 \leq d_{ip} \leq 1, \ \forall i, p \tag{5}$$

Equation 6 states that each entity is mapped onto one and only one partition.

$$\sum_p x_{jp} = 1, \ \forall j \tag{6}$$

Based on equation 7, the switching cost of all entities in each row of a partition has to contain only 0 values in order to be able to compressed and thus removed from the configuration bitstream.

$$\sum_j (x_{jp} \cdot v_{ij}) - N(1 - d_{ip}) \leq 0, \ \forall i, p \tag{7}$$

We use a commercial ILP solver [29] to perform the computations.

6. Evaluation

6.1 Experimental Setup

We test the proposed method on a commercial CGRA, the Samsung Reconfigurable Processor [27]. The processor consists of 16 PEs, 12 register files, 8 constant units, and a large number of multiplexers to control the interconnection network. A configuration line for the total 383 configurable entities is 1280 bits wide.

The compression scheme has been implemented in the proprietary Samsung C Compiler. We have modified the compiler toolchain to generate temporally optimized code.

Application class	Applications (# kernels)	Total kernels	Total conf.lines
Graphic	3D (9) matrix (3) opengles_r269 (9) PICKLE_V1.2 (14)	35	155
Video	aac.1 (16) avc.swo (11) EaacPlus.1 (23) mp3.1 (10) mpeg_surround(26)	86	588
Voice	FIR (1) huffman_decode (6) bit_conversion (1) histogram (1) amr-wbPlus (12)	21	174
Voice (SIMD)	FIR (1) high_pass_filter (1) huffman_decode (5) bit_conversion (1) histogram (1)	9	116
Resolution (SIMD)	bilateral (3) gaussian_smoothing (3) optical_transfer_function (6)	12	101
Imaging (SIMD)	csc (1) dct (1) median (1) sad (1)	4	73
Others	word_count (13) merge_sort (5) bubble_sort (3) array_add (5)	26	210
Total	30	193	1417

Table 1. Application benchmarks.

The logical partitions of the configuration memory are optimized for different setups.

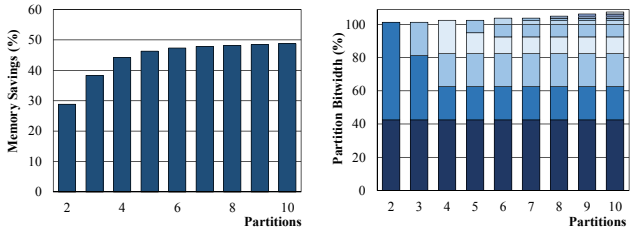
The overhead of the partitioned configuration memory and the necessary decoder logic in terms of area, power, and timing information has been synthesized using the Synopsis Design Compiler [28]. For the configuration memory energy computation we use CACTI 6.5 [18].

6.2 Benchmarks

The benchmarks used for the evaluation are thirty real-world applications deployed in smartphones, cameras, printers, and other high-end mobile devices. The applications contain a total of 193 loop kernels. CGRAs often target at a specific application domain at design-time; to reflect this we have grouped the applications into seven application domains. Groups with the `(SIMD)` suffix use customized instructions for the respective application domain. Table 1 lists the application domains and benchmark applications.

6.3 Effect of Partitioning

The results of partitioning all loop kernels from two to 10 partitions using the edit-distance heuristics are shown in Figure 9. The memory savings are shown on the left, the width of each partition in relation to the original configuration width on the right-hand side of the figure. More partitions



(a) (b)

Figure 9. Memory reduction and partition bitwidths in dependence of the number of partitions.

allow for higher memory savings thanks to the increased flexibility when grouping entities, but lead to overhead in terms of the decoder logic and padding of the individual partitions up to the next multiple of 16 bits. The increase of the total bitwidth becomes noticeable at six partitions or more. We observe that the potential memory savings are significant even with a relatively low number of partitions. In the following, we use a four-partition layout for all experiments.

6.4 Compression and Energy Reduction

Next, we compare the performance of the heuristic with the GA and the ILP solution in terms of computational complexity, memory reduction, and runtime energy reduction. The ILP solver failed to find a solution for all but individual loops. A comparison of the performance in terms of memory reduction of the heuristics, the GA, and the ILP for the individual loops shows that, on average, the ILP outperforms the GA by 1 and the heuristics by 2 percent.

The results per application domain for the heuristics and the GA are summarized in Table 2. For the heuristics, the results of one run are reported (the algorithm is deterministic). The GA is run twenty times, the best result is reported for the memory and energy reduction, and the CPU time shows the total time of the 20 runs. Overall, both the heuristics and GA achieve compression rates between 28 to 56% and a reduction in the energy consumed by the configuration memory of 22 to 47 percent. The overhead of the decoder logic amounts to 2-3% of the original configuration memory energy. We observe that the GA achieves slightly better results at the expense of a significantly increased computational complexity. The results suggest that the heuristics based on edit-distance is a good candidate for the architecture exploration phase where quick results are desired, and once the system design has been fixed, the GA is used to compute the final partitioning of the configuration memory.

Looking at the different application domains, we observe that the performance for SIMD-based applications falls behind that of non-SIMD applications. By regrouping applications we found that the main reason for this outcome is mostly due to the limited number of configuration lines available for the statistical analysis and caused by patterns specific to the application domain.

Application class	Heuristics			Genetic algorithm		
	Memory Reduction (%)	Runtime Energy Reduction (%)	CPU Time (s)	Memory Reduction (%)	Runtime Energy Reduction (%)	CPU Time (s)
All loops	44	36	26.6	47	38	30,822
Graphic	53	45	3.9	56	47	1,275
Video	53	43	14.0	56	45	7,248
Voice	48	40	3.9	53	44	2,971
Voice (SIMD)	43	35	2.8	49	40	2,340
Resolution (SIMD)	28	22	2.5	30	23	1,256
Imaging (SIMD)	34	27	1.9	40	31	892

Table 2. Computational complexity, memory reduction, and runtime energy reduction for the heuristics and the genetic algorithm for different application domains.

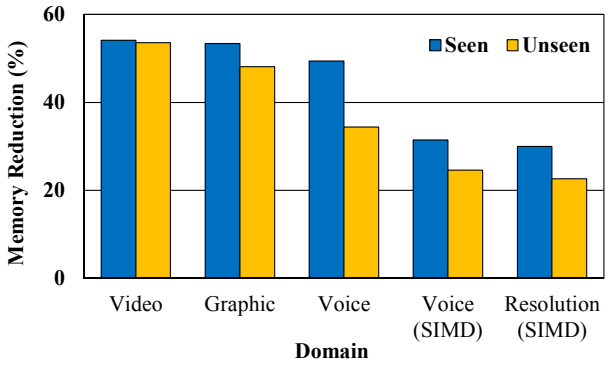


Figure 10. Memory reduction per application domain for seen and unseen applications.

6.5 Compressibility of Seen vs. Unseen Code

A big advantage of CGRAs is their reconfigurability after shipment. 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 new code. We are thus interested in the compressibility of unseen code from a specific application domain.

For the test, we use a 80:20 ratio, that is, 80% of the loops are assumed to be known and fed into the statistical analyzer to compute a partitioning. The remaining 20% are then compiled with the fixed partitioning. The results for the application domains that allow a 80:20 partitioning are shown in Figure 10. We observe good compressibility even for loops that have not been part of the training set; for most application domains the loss in compressibility is less than 10%. Only *Voice* drops by about 15%; this is because the applications in that domain exhibit a larger variation in the code patterns than the other domains.

7. Conclusion

In this paper, we presented an efficient code compression/decompression technique for coarse-grained reconfigurable architectures. The technique is able to significantly reduce the amount of memory required to store a modulo schedule's configuration as well as the energy consumed by the configuration memory. Compression is achieved by removing identical consecutive configuration lines in the schedule. Compressibility is improved by an post-pass optimization that minimizes the number of signal changes in the temporal dimension. In addition, we split the configuration lines into several partitions and allocate signals with similar change patterns into the same partition. Decompression is guided by a bitvector that guides a simple hardware decompression logic. The proposed technique has been implemented in a production-level modulo scheduler for the Samsung Reconfigurable Processor and evaluated on a 4x4 architecture with a wide range of loop kernels from real-world applications. For different application domains, the proposed method achieves space savings between 28 and 53 percent for different application domains. At runtime, the configuration memory energy consumption is reduced by 22 to 45%.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions. This work was supported in part by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program through 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.

References

[1] N. Aslam, M. J. Milward, A. T. Erdogan, and T. Arslan. Code compression and decompression for coarse-grain reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(12):1596–1608, Dec 2008.

[2] M.-K. Chung, Y.-G. Cho, and S. Ryu. Efficient code compression for coarse grained reconfigurable architectures. In *IEEE 30th International Conference on Computer Design (ICCD)*, pages 488–489, 2012.

[3] M.-K. Chung, J.-K. Kim, Y.-G. Cho, and S. Ryu. Adaptive compression for instruction code of coarse grained reconfigurable architectures. In *International Conference on Field-Programmable Technology (FPT)*, pages 394–397, 2013.

[4] J. H. Holland. Adaptation in natural and artificial systems. *MIT Press, Cambridge, MA*, 1992.

[5] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim. ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications. In *International Conference on Field-Programmable Technology (FPT)*, pages 329–334, 2012.

[6] Y. Kim, I. Park, K. Choi, and Y. Paek. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 310–315, 2006.

[7] Y. Kim, R. N. Mahapatra, I. Park, and K. Choi. Low power reconfiguration technique for coarse-grained reconfigurable architecture. *IEEE transactions on very large scale integration (VLSI) systems*, 17(5):593–603, 2009.

[8] T. Kitaoka, H. Amano, and K. Anjo. Reducing the configuration loading time of a coarse grain multicontext reconfigurable device. In *International Conference on Field Programmable Logic and Applications*, pages 171–180, 2003.

[9] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, 1988.

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

[11] J. Lee, Y. Shin, W.-J. Lee, S. Ryu, and K. Jeongwook. Real-time ray tracing on coarse-grained reconfigurable processor. In *International Conference on Field-Programmable Technology (FPT)*, pages 192–197, Dec 2013.

[12] W.-J. Lee, S.-H. Lee, J.-H. Nah, J.-W. Kim, Y. Shin, J. Lee, and S.-Y. Jung. SGRT: a scalable mobile gpu architecture based on ray tracing. In *ACM SIGGRAPH 2012 Posters*, page 44, 2012.

[13] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han. SGRT: a mobile gpu architecture for real-time ray tracing. In *Proceedings of the 5th high-performance graphics conference*, pages 109–119, 2013.

[14] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, H.-S. Park, S. Jung, and S. Lee. A novel mobile gpu architecture based on ray tracing. In *2013 IEEE International Conference on Consumer Electronics (ICCE)*, pages 21–22, 2013.

[15] B. Liu, W.-Y. Zhu, Y. Liu, and P. Cao. A configuration compression approach for coarse-grain reconfigurable architecture for radar signal processing. In *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 448–453, Oct 2014.

[16] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *13th International Conference on Field Programmable Logic and Applications (FPL)*, pages 61–70, 2003.

[17] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings - Computers and Digital Techniques*, 150(5):255–261, Sept 2003.

[18] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[19] T. Oh, B. Egger, H. Park, and S. Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 21–30, 2009.

[20] Y. Park, H. Park, and S. Mahlke. Cgra express: Accelerating execution using dynamic operation fusion. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 271–280, 2009.

[21] M. Quax, J. Huisken, and J. van Meerbergen. A scalable implementation of a reconfigurable wcdma rake receiver. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE) - Volume 3*, pages 30230–, 2004.

[22] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *27th Annual International Symposium on Microarchitecture (MICRO)*, pages 63–74, 1994.

[23] Samsung Exynos 4210 Product Brief. `http://www.samsung.com/us/business/oem-solutions/pdfs/Exynos\_v11.pdf`, 2011. (online; accessed November 2016).

[24] K. Sastry, D. E. Goldberg, and G. Kendall. Genetic algorithms. In *Search methodologies*, pages 93–117. Springer US, 2014.

[25] Y. Shin, J. Lee, W.-J. Lee, S. Ryu, and J. Kim. Full-stream architecture for ray tracing with efficient data transmission. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2165–2168, 2014.

[26] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers*, 49(5):465–481, 2000.

[27] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim. Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In *International Conference on Field-Programmable Technology (FPT)*, pages 67–70, 2012.

[28] Synopsys Design Compiler 2010. `http://www.synopsys.com/`, 2010. (online; accessed November 2016).

[29] TOMLAB CPLEX. `http://www.tomopt.com/tomlab/products/cplex`, 2016. (online; accessed November 2016).

[30] V. Tunbunheng, M. Suzuki, and H. Amano. Romultic: Fast and simple configuration data multicasting scheme for coarse grain reconfigurable devices. In *IEEE International Conference on Field-Programmable Technology*, pages 129–136, 2005.

A. Artifact Description

A.1 Abstract

The code compression techniques presented in this paper are implemented in C++ directly in the proprietary Samsung C Compiler (SCC) for the Samsung Reconfigurable Processor (SRP). In order to support the artifact evaluation process, we have extracted the related code and re-implemented the full functionality with a number of Python scripts. For the evaluation with Genetic Programming, a simple C-based implementation of a GA is provided. The ILP formulation requires Matlab and the TOMLAB/CPLEX solver; for licensing reasons we cannot provide these programs with the artifact package, however, the results can be verified using evaluation licenses.

The raw benchmark data of the 193 loop kernels is available in pre-compiled form, i.e., we provide the original, uncompressed CGRA schedules as generated with the original Samsung C compiler. Using the included scripts and detailed instructions inside the AE package, all results in the paper regarding compressibility, configuration memory partitioning, and energy savings for single, all, or application domains as outlined in the paper can be verified.

All of the above is provided in a VirtualBox image. By following the instruction contained in the AE package itself, interested readers are able to examine the source code of the heuristics and the genetic algorithm, reproduce the results given in the paper, run and verify the test cases for the different algorithms, or compose new combinations of loop kernels and examine the results.

A.2 Description

A.2.1 Check-list (Artifact Meta Information)

- **Algorithms**: provided are algorithms for resource sorting based on edit distance, asap-alan temporal optimization, temporal optimization based on genetic programming, and the integer linear programming formulation.
- **Program sources**: the source code of the algorithms are implemented in Python, C, and Matlab. The driver programs are written in Bash script. All necessary software to compile and run the algorithms is included *except* for the ILP (Matlab/TOMCAT must be obtained separately).
- **Run-time environment**: the algorithms run on any platform, provided the necessary compilation/runtime tools are available. The provided AE VM image runs Linux.
- **Hardware**: no dependencies.
- **Input data set**: the input data are the original, unmodified modulo schedules of 193 loop kernels from 30 real-world applications from different application domains. The schedules were generated by the proprietary Samsung C compiler. The schedules are stored in a (human-readable) textual format and included in the AE package.
- **Output data set**: generated are a partitioning scheme for a given set of benchmarks along with the compressed modulo schedules of the loop kernels as configuration memory data. The output uses the same human-readable textual format as the input. In addition, statistics about the partitioning, compressibility, and runtime energy savings are produced.
- **Experimental workflow**: Download the VirtualBox AE machine, run experiments in VM, compare results.
- **Publicly available?** available upon request. Send email to `bernhard@snu.ac.kr`.

A.2.2 How Delivered

The artifact is provided in form of a VirtualBox VM image running Ubuntu 16.04 LTS. The image includes all the necessary input data, scripts, Python and C sources, and technical documentation to run, and verify the results presented in the paper. Users interested in obtaining access to the artifact are welcome to contact us at `bernhard@snu.ac.kr`.

A.2.3 Hardware Dependencies

There are no hardware dependencies. The Genetic Algorithm and the ILP have a high computational complexity and work better on machines with multiple processors and a sufficient amount of RAM.

A.2.4 Software Dependencies

The ASAP-ALAN algorithm and the edit-distance heuristics is implemented in Python 3, while the GA is implemented in C. The driver scripts are written for the GNU Bash shell or Python 3. All necessary tools have been preinstalled on the VM and are ready to use. The notable exception is the ILP formulation which is provided as a Matlab program. As a consequence, Matlab and the commercial TOMCAT/CPLEX solver are required to run the ILP. Installation instructions are provided below.

A.2.5 Datasets

We provide the uncompressed modulo schedules for a commercially used CGRA, the 4x4 SRP, as generated by the unmodified Samsung C Compiler for SRP in textual (CSV) format for a total of 193 loop kernels from 30 real-world applications. Table 2 and Figure 10 in the paper show compressibility and energy savings for different groups of loop kernels to reflect the fact that the SRP is typically used in specific application domains such as smartphones, cameras, or TV sets. For easy reproduction the same input data groups are provided in the artifact image. In addition, scripts are provided to reproduce the results of Table 2 with two simple commands.

The format of the input and output data is a human-readable textual format. The modulo schedules of the loop kernels list the different reconfigurable hardware entities and their configuration signals. Information about the partition-

ing of the configuration memory is also provided in textual format, listing all entities that are grouped together in a partition. In addition to the raw output data, detailed information about the compressibility and energy consumption for the different benchmark groups and the three evaluated techniques is provided. Note that the ILP solver, due to the complexity of the problem space, can only compute a solution for single loops (as outlined in the paper in Section 6.4).

A.3 Installation

The provided AE package first needs to be imported into VirtualBox (`https://www.virtualbox.org/`). The default settings (2 CPUs, 2 GB of RAM) suffice to run the heuristics and the GA and solve the ILP for simple loops.

For the edit distance and the GA heuristics, all necessary tools have been pre-installed in the VM. The ILP solver requires Matlab and the commercial TOMCAT/CPLEX solver. Due to licensing problems, we are unable to include these software packages in the VM, however, for both packages, trial versions are available online:

- **Matlab**
  `https://www.mathworks.com/programs/trials/trial_request.html?s_tid=gn_loc_drop`
- **TOMLAB/CPLEX**
  `http://www.tomopt.com/tomlab/products/cplex/`

Download and install the packages, then follow the instructions in the ILP subdirectory. Note that the ILP solver requires a significant amount of computation and memory to run. If the computation aborts or does not complete in the provided VM image, we recommend running it on a multicore platform with sufficient RAM.

A.4 Experimental Workflow

The AE package includes a series of shell scripts that perform the experiments described in the paper and generate the results of each memory compression technique. Users can easily reproduce the experiments by following the steps below.

- Obtain a link to the AE VM image
  (contact `bernhard@snu.ac.kr`).
- Download and import the VM image into VirtualBox.
- Start the VM, you are automatically logged in. Open the README.html file on the Desktop and follow the instructions provided inside the AE package itself.
- The results are generated in separate subdirectories for the different techniques. Detailed explanations regarding the format of the generated results are provided inside the AE VM.

A.5 Evaluation and Expected Result

The generated results include the compressed configuration code of the loop kernels and detailed statistics listing partitioning information, compressibility, and energy savings for the input data set (single loops or groups of loops). Except for the runtimes, the results should be identical to those reported in the paper. The runtime of the different algorithms heavily depends on the underlying hardware and can thus vary significantly.

A.6 Artifact Evaluation Methodolody

Details about the submission and reviewing methodology of the artifact can be found at the following address:
`http://ctuning.org/ae/submission-20161020.html`