A Bimodal Scheduler for Coarse-Grained Reconfigurable Arrays

PANAGIOTIS THEOCHARIS and BJORN DE SUTTER, Ghent University, Belgium

Compilers for Course-Grained Reconfigurable Array (CGRA) architectures suffer from long compilation times and code quality levels far below the theoretical upper bounds. This article presents a new scheduler, called the Bimodal Modulo Scheduler (BMS), to map inner loops onto (heterogeneous) CGRAs of the Architecture for Dynamically Reconfigurable Embedded Systems (ADRES) family. BMS significantly outperforms existing schedulers for similar architectures in terms of generated code quality and compilation time. This is achieved by combining new schemes for backtracking with extended and adapted forms of priority functions and cost functions, as described in the article. BMS is evaluated by mapping multimedia and software-defined radio benchmarks onto tuned ADRES instances.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms: Architecture, Compiler, Performance

Additional Key Words and Phrases: Modulo scheduling, placement and routing, cost functions

ACM Reference Format:
Panagiotis Theocharis and Bjorn De Sutter. 2016. A bimodal scheduler for coarse-grained reconfigurable arrays. ACM Trans. Archit. Code Optim. 13, 2, Article 15 (June 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2893475

1. INTRODUCTION

Since around 2000, Coarse-Grained Reconfigurable Array (CGRA) architectures have been proposed to accelerate inner loops of Digital Signal Processing (DSP) applications [De Sutter et al. 2013]. Extensions to CGRAs such as Single Instruction, Multiple Data (SIMD) support, virtualisation, dynamic operation fusion, custom memory configurations, and multi-threading allow many forms and levels of parallelism to be exploited [Park et al. 2013, 2009b, 2009c; Jang et al. 2011; De Sutter et al. 2010; Kanstein et al. 2007; Vander Aa et al. 2011; Pager et al. 2015].

One type of CGRA, named Architecture for Dynamically Reconfigurable Embedded Systems (ADRES), can be seen as a coarse-grained Field-Programmable Gate Array (FPGA) in which look-up tables have been replaced by word-wide Functional Units (FUs) and Register Files (RFs) and in which a new configuration for the whole CGRA is loaded every cycle [Mei et al. 2003]. ADRES can also be seen as an extension of clustered Very Large Instruction Word (VLIW) architectures to a more generic, two-dimensional type of VLIW. ADRES CGRAs are, for example, also scheduled statically. However, whereas each FU in a VLIW is typically connected to only a single RF via

This work was supported by the Agency for Innovation by Science and Technology in Flanders, Belgium, under Grant No. 101605. This work was also supported by the concerted research action BOF11/GOA/021 of Ghent University.
Authors' addresses: P. Theocharis and B. De Sutter, Computer Systems Lab, Ghent University, Technologiepark-Zwijnaarde 15, 9052 Gent, Belgium; emails: panagiotis.theocharis@ugent.be, bjorn.desutter@ugent.be.

DOI: http://dx.doi.org/10.1145/2893475

multiple implicit connections and ports dedicated to that FU, ADRES FUs are typically connected to multiple, distributed RFs and to other FUs over connections that are shared. Because of that feature, programming ADRES CGRAs remains a challenge. On a VLIW, scheduling an operation in some slot implies the reservation of many resources, including an FU and its dedicated connections and ports. This simplifies the code generation, which can rely on relatively simple reservation tables during modulo scheduling to generate software-pipelined loop code. Because individual ports and connections in a CGRA are shared, they have to be allocated explicitly at compile time and programmed explicitly at run time. The compiler hence not only needs to allocate slots (i.e., FU-cycle pairs) to schedule operations, it also needs to allocate the necessary resources to transfer data between slots. The task of an ADRES compiler back-end hence consists of Placement & Routing (P&R), two terms borrowed from the domain of FPGAs. Operations need to be placed on the computational resources in the CGRA schedule, and data dependencies need to routed over the interconnect resources.

P&R tasks typically require much more compilation time than traditional VLIW code generation. This problem is aggravated by the fact that many features can be customized per individual resource. Design space exploration studies have shown that this customizability is advantageous for targeting different application domains [Novo et al. 2009; Oh et al. 2009; Lambrechts et al. 2009; Cervero et al. 2008; Bouwens et al. 2008]. Actual ADRES hardware prototypes and commercialized instances (named Samsung Reconfigurable Processor) designed and fabricated for different application domains confirm this observation for domains such as Multimedia (MM) [Mei et al. 2008], Software-Defined Radios (SDRs) [Bougard et al. 2008; Derudder et al. 2009; Suzuki et al. 2011], and audio processing [Suh et al. 2012]. It is therefore advantageous for ADRES compilers to be retargetable to different, heterogeneous ADRES instances.

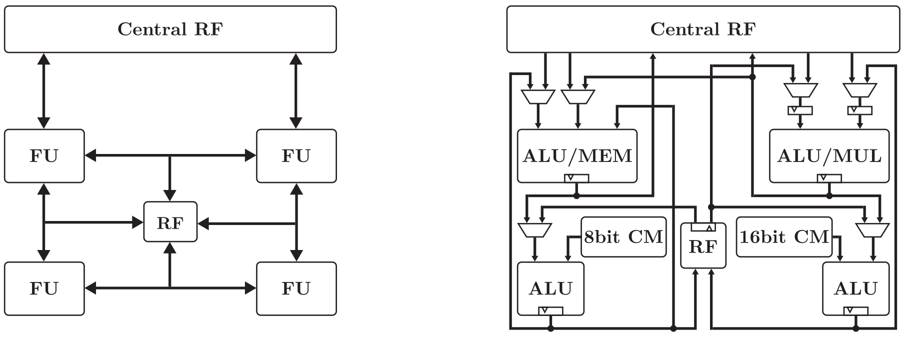
Many modulo schedulers have been proposed for ADRES-like CGRAs [Mei et al. 2002; De Sutter et al. 2008; Park et al. 2008; Oh et al. 2009; Lee et al. 2011; Hamzeh et al. 2012, 2013; Kim et al. 2013; Chen and Mitra 2014]. All of them offer a specific tradeoff among generated code quality, compilation time, and supported features and heterogeneity in the targeted CGRAs. None of the existing algorithms reaches the theoretical code quality limit, however, and the ones that come close are slow, requiring up to tens of seconds to schedule a single (unrolled) inner loop body. Clearly, further improvements are needed, for example, to support automatic CGRA design space exploration, which requires code to be generated for many CGRA instances. This article pushes the state of the art with the following contributions:

(1) This article presents a BMS that extends and improves concepts from existing algorithms in several ways. It achieves its goal by combining new forms of backtracking, new priority functions, and new cost functions.
(2) We present different ways to configure the BMS, that is, to deploy its two modes. This allows the developer to trade off code quality and compilation time.
(3) Building on the flexible hardware resource models from the original Dynamically Reconfigurable Embedded Systems Compiler (DRESC), the BMS supports the widest range of heterogeneous ADRES instances, while requiring no more compilation time than compilers that target more homogeneous instances.

Section 2 provides background information and related work. Section 3 describes the BMS, which is evaluated in Section 4. Conclusions are drawn in Section 5.

2. BACKGROUND

The design space of CGRA architectures is vast and features many design options [De Sutter et al. 2013]. Our research focuses on ADRES [Mei et al. 2003], a template for CGRAs coupled tightly to a main Central Processing Unit (CPU). A simplified ADRES instance is depicted in Figure 1(a). It has a large central RF, four FUs, and one local RF.



(a) Abstract view on a 2×2 ADRES CGRA (b) Concrete, heterogeneous 2×2 ADRES

Fig. 1. Two views on the same 2×2 ADRES instance.

When executing code outside inner loops, the processor operates as a VLIW processor, in which only the central RF and the top row of FUs are active.

When executing inner loops, the whole CGRA is active in the so-called CGRA mode. In each cycle, all elements of the array are then configured by a very wide configuration word read from a configuration memory. To minimize the energy overhead of this per-cycle reconfiguration, no control flow is supported in CGRA mode, and it is scheduled completely statically. Alternatives to a wide configuration memory have been proposed in the literature [Park et al. 2009a], but as those techniques tackle the control path implementation of a CGRA, they are orthogonal to the code scheduling problem tackled in this article, which is in essence a data path control problem. To generate control-flow-free loop bodies, predication is used [Mahlke et al. 1992].

Figure 1(b) shows a more detailed picture of the same ADRES instance of Figure 1(a), in which many of the elements that need to be configured are visible: All ports to RFs need to be configured with read and write addresses, all Multiplexers (MUXs) need to be configured to select the appropriate inputs, and all FUs need to be configured to execute the scheduled operation. In our ADRES designs, in each cycle each so-called Constant Memory (CM) outputs a (small) constant value to be routed to the FUs that need constant operands. The constant to be generated each cycle needs to be "configured" as well. Alternative solutions for storing constants exist, such as pre-loading them into RFs or accessing them from local RAMs connected to FUs. Those alternatives are not targeted in this article, however.

Switching from VLIW to CGRA mode takes a couple of cycles. Data are passed from non-loop to loop code via the central RF and the memories that the Load/Store (LD/ST) units can access in both modes [Jang et al. 2011; De Sutter et al. 2010].

An important aspect of ADRES is the heterogeneity supported by the architecture template, as shown in Figure 1(b). Per individual FU, a designer can customize the pipeline depth, the number and widths of operands, as well as support for custom instructions such as SIMD and application-specific instruction set extensions, and for generic Arithmetic Logic Unit (ALU), Multiplication (MUL), and LD/ST operations. For each RF, the number of registers, their width, and the number of read and write ports have to be chosen. For each CM, the width of the supported constants has to be decided. All connections among the FUs, RFs, CMs, and MUXs can be specified individually and pipeline latches can be placed anywhere in the interconnect network to obtain the best combination of static schedule lengths and clock frequency, and to trade off performance and energy consumption. Numerous design exploration experiments have demonstrated that it is useful to explore all of these forms of heterogeneity [Mei et al.

```
 1 int example()
 2 {
 3  int i = 0;
 4  int j = 0;
 5  int k = 0;
 6
 7  for(;i<512;i++)
 8  {
 9   int l = j + 2;
10   int m = k + 4;
11   j = l + m * 3;
12   k = m - l * 3;
13  }
14
15  return i + j + k;
16 }
```

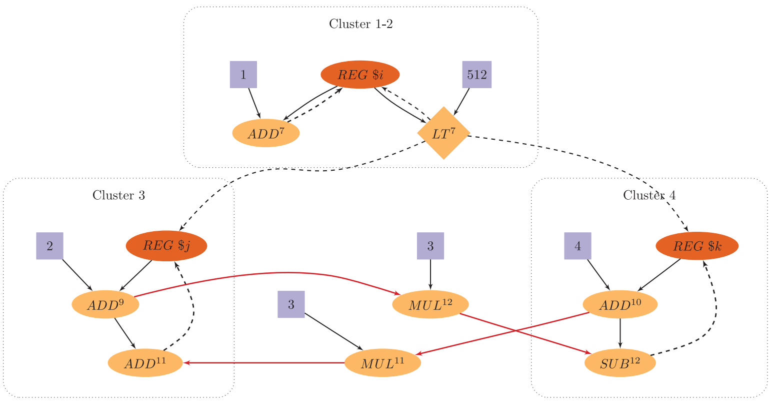


Fig. 2. Code example and its DDG. Operations in the DDG are annotated with the line number, live-in/live-out registers are annotated with the corresponding variable name.

2008; Lambrechts et al. 2009; Novo et al. 2009; Oh et al. 2009] and hence to consider support for heterogeneity when designing or evaluating code generation techniques.

2.1. Code Generation Based on Simulated Annealing

Almost simultaneously with ADRES itself, its inventors published compiler techniques implemented in a compiler named DRESC [Mei et al. 2002, 2003]. Following multiple improvements over several years [De Sutter et al. 2008], DRESC still generates (close to) the best quality code, albeit at very long compilation times.

DRESC's modulo scheduler maps a Data Dependence Graph (DDG) representation of the loop body onto a Routing Resource Graph (RRG) representation of the CGRA. An example of such a DDG is depicted for its corresponding source code in Figure 2. The DDGs nodes represent operations (yellow ovals), constant operands (purple squares), and live-in and live-out registers (orange ovals). The latter are registers through which variables' values are passed from the VLIW mode (that executes the code before and after the loop) to the CGRA mode (that executes the loop iterations) and back. The DDG edges model data dependencies. The RRG is a directed graph representation borrowed from FPGA P&R techniques [Ebeling et al. 1995]. In the RRG, each hardware resource is modelled with a node or set of nodes in every cycle. RRG edges model connections between these resources. Edges in the RRG from a node in one cycle to a node n cycles later model that the resource at the tail of the edge has a latency of n cycles. Each DDG node corresponding to an operation, live-in/out register, or constant is *placed* on an RRG node of the corresponding type in some cycle of the schedule. The DDG edges are mapped, or *routed*, onto so-called nets through the RRG, that is, they are routed over RRG paths containing nodes that model MUXs, latches, FU and RF ports, registers in the RFs, and so on. An advantage of DRESC's approach is its support for heterogeneity: By adding additional nodes that model virtual hardware resources to the RRG and by connecting them appropriately, a wide range of FUs, RFs, and interconnect designs can be modelled as discussed in literature [Mei et al. 2002; De Sutter et al. 2008].

Compilers like DRESC invoke a modulo scheduler at increasing Initiation Intervals (IIs) until they find a valid schedule. Lower IIs are better because in a modulo schedule with some II, a new iteration of the loop body is initiated every II cycles. In such a schedule, an operation placed on a resource in some cycle c occupies that resource in all cycles c' for which c' mod II $= c$ mod II. To model this during the scheduling, the RRG is augmented with concepts borrowed from modulo reservation tables [Lam 1988; Rau 1994] into a so-called modulo RRG or MRRG. The Minimum Initiation Interval

(MII) is determined by the recurrent data dependencies in the loop and by the number of computational resources required by the loop relative to the resources available in the CGRA. MII is computed as MAX(RecMII, ResMII). RecMII is the minimal schedule length of the largest recurrent data dependency chain in the loop's DDG. ResMII is the minimal number of cycles needed to get enough FU slots to schedule all the loop's FU operations. The MII puts a lower bound on the obtainable II. Hence the compiler performs its first modulo scheduling attempt for an II value of MII.

Per the attempt at some II, DRESC first starts with generating a randomized, invalid schedule that overuses resources. In this initial schedule, multiple DDG nodes can be placed on the same MRRG node and multiple DDG edges can be routed over the same MRRG nodes. For that reason, generating those schedules is easy and done very quickly. An MRRG node is overused $n - 1$ times if it is used for n DDG nodes or edges.

Then an iterative repair phase starts, in which the compiler tries to replace operations, that is, move them from one node in the MRRG to another, until all overuse of nodes is eliminated and a valid schedule is obtained. To find good replacements for an operation, the compiler tries several randomly selected new places, of which it selects the one with the lowest routing cost. A place's routing cost includes the sum of the overuses of all nodes on the nets used to route the incoming and outgoing DDG edges of the operation being replaced. For each place, the cheapest nets are determined with a cheapest path router that also includes overuse in its cost function. So from all tried replacements, the place is selected that reduces the total overuse of all MRRG nodes the most. When the total overuse reaches zero, a valid schedule is found. When it does not reach zero, and the tried replacements make no more progress towards zero, the attempt is aborted, and the scheduler tries again at a higher II. To avoid that the overuse minimization gets trapped in local minima, Simulated Annealing (SA) is used.

This approach suffers from two major drawbacks. First, it is terribly slow. The cheapest path algorithm that forms the inner loop is executed so many times that even generating code for a single loop can take minutes and even tens of minutes. Second, the approach has quite a high chance of not finding the best schedules. In order to avoid an explosion in the number of cheapest routes to be computed, DRESC only considers one DDG operation for replacement at a time. In other words, each individual replacement of an operation needs to be accepted by the SA algorithm. When at some point during the SA a lower total overuse can only be reached by replacing a whole group of operations from one side of the CGRA to another, DRESC will still get stuck in a local minimum. Whether or not that happens depends on the seed used to initialize the Pseudo-Random Number Generator (PRNG) used by DRESC to generate the initial randomized schedule. So to ensure that the compiler misses no valid schedules at low IIs, the user has to re-invoke the scheduler with different random seeds. Alternatively, the user can invoke multiple attempts in parallel. But since even a single attempt to generate a valid schedule is slow, this will still not yield fast compilation.

2.2. List Schedulers

As a faster alternative, several list schedulers have been developed for ADRES-like architectures. They also try to generate valid schedules at increasing IIs, but, like traditional list schedulers, they start each scheduling attempt with an empty schedule, in which they then iteratively place operations one by one. They route dependencies as soon as both involved nodes are placed and never introduce any overuse. Most list schedulers feature no or minimal backtracking. This allows them to be fast, but it can also prevent them from finding the best schedules because it is quite common to make wrong placement decisions early that only much later turn out to cause problems.

Edge-centric Modulo Scheduler (EMS) and its successor, Recurrence cycle Aware Modulo Scheduler (RAMS), are two routing-based list schedulers [Park et al. 2008; Oh

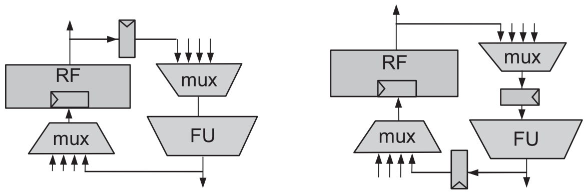


Fig. 3. Pipelined interconnects of our multimedia (left) and SDR (right) CGRAs.

et al. 2009]. Suppose the n nodes in a DDG are numbered from 1 to n in the order in which the list scheduler's *priority function* will try to place them in the schedule. Each time a node i from the DDG is to be placed, these schedulers determine the set PP_i of all potential places to which all data dependencies from the already-placed nodes $A_i = \{1, \ldots, i-1\}$ to node i can still be routed over the set NR_i of resources not reserved yet. The set PP_i is typically bound by the type of operation of node i and its ALAP and ASAP (As Soon/Late As Possible) times that bound the cycles in the static schedule in which the operation can be placed. From PP_i the scheduler then picks the one place that minimizes the cost of routing those dependencies to node i.

While scheduling node i and routing its dependencies from nodes in A_i, the cost $C_{i,j}$ of using a resource j reflects the estimated impact the reservation of j for i will have on future opportunities to find valid places for the nodes in $P_i = \{i+1, \ldots, n\}$ that will need to be placed later and on the future opportunities to find valid routes for dependencies from already placed nodes in A_i to yet-to-be-placed nodes in P_i. In other words, the $C_{i,j}$ is an estimate of the importance to keep resource j available for nodes in P_i, given the placement of nodes in A_i. For example, consider a situation in which an ALU operation i is being placed, P_i contains three LD/ST operations, and only three FU slots remain available in the schedule for executing those three LD/ST operations. Those three slots will be assigned a large cost to try to avoid that ALU operation i occupies them.

The P&R performed by these list schedulers can hence be summarized as a process that tries to look ahead to ensure that all operations in P_i not scheduled yet and all data dependencies not routed yet will find a valid place and net in the future. Park et al. presented a set of *cost functions* ranging from very simple to quite complex to be aggregated into $C_{i,j}$ for EMS. Those cost functions were reused in RAMS.

While being one to two orders of magnitude faster than DRESC, these schedulers still require multiple seconds to generate code for a non-trivial loop. The reason is that $C_{i+1,j} \neq C_{i,j}$ for most j because $P_{i+1} \neq P_i$, $A_{i+1} \neq A_i$, and $R_{i+1} \neq R_i$. In other words, after every placement of a node i, considerable effort is needed to recompute the different cost functions' contributions to $C_{i+1,j}$ for all relevant j. As a result, there is still a lot of room, and a need, for improving the existing routing-based list schedulers.

The main differences between EMS and RAMS are their priority functions and backtracking strategies. RAMS clusters nodes in recurrence cycles in the DDG and tries to place cluster by cluster, assigning higher priority to clusters whose recurrence lengths are dominant, that is, closest to the II being attempted. The reason is that, by definition, all nodes in a recurrence cycle in the DDG need to be placed within a window of at most II cycles in the schedule [Llosa et al. 2001]. Those nodes are hence most constrained and are therefore scheduled as soon as their predecessors (i.e., the so-called *incoming tree* of the recurrence cycle) have been scheduled. For RAMS, code qualities similar to those of DRESC are reported, and even slightly better.

EMS and RAMS are evaluated on CGRAs described more at the abstraction level of Figure 1(a) than that of Figure 1(b). For example, the insertion of pipeline latches in the interconnect (as exemplified in Figure 3) is not mentioned in those articles, while their importance is clear from several design experiments [De Sutter et al. 2013; Mei et al.

2008; Kim et al. 2005; Lambrechts 2009]. Moreover, with the exception of some FUs supporting more instruction classes than others and different classes of distributed RFs, EMS is evaluated on rather homogeneous CGRAs. The target used to evaluate RAMS shows only a bit more heterogeneity, as Oh et al. also present an architecture improvement in the form of dedicated, heterogeneously connected RFs.

Several years after EMS and RAMS were published, Kim et al. identified some supposed shortcomings with respect to those algorithms' scheduling of mutually dependent recurrence cycles in a DDG [Kim et al. 2012]. As an extension, they presented the Strongly Connected Components-based Modulo Scheduler (SCCMS). On top of clustering recurrence cycles, SCCMS detects the SCCs that identify mutually dependent clusters and takes them into account in its priority functions. Kim et al. claim to generate much better code than RAMS as a result, with some loops being up to a factor 3.5 faster. They also claim to be able to schedule more loops in the first place, because SCCMS allegedly does not deadlock when RAMS sometimes does. These results are obtained only for pathological loops that feature some properties that make them hard to schedule for RAMS but that would pose no problems for DRESC. In fact, in our years of experience with DRESC, we never observed any loop being scheduled at an II that comes even close to 3.5 times the lower bound of the loop's II, that is, the loop's MII.

Before EMS and RAMS were proposed, the same research group proposed a technique called Modulo Graph Embedding (MGE) [Park et al. 2006]. MGE uses similar heuristics (and cost functions) as the routing-based schedulers for finding good places for operations, but it uses a simplified resource model that only supports dedicated point-to-point connections. It, hence, does not support the type of ADRES instances targeted in this article, which include shared connections.

Several other graph-based schedulers have been proposed that build heavily simplified resource graphs to model the CGRA [Yoon et al. 2008; Hamzeh et al. 2012, 2013; Chen and Mitra 2014]. Using different customized algorithms to find limited forms of sub-graph isomorphisms between a loop's DDG and the architecture resource graph, these schedulers can generate schedules very quickly. However, the limitation to certain forms of sub-graph isomorphisms can result in significantly lower code quality. Moreover, the simplified resource graphs cannot express many kinds of heterogeneity and features, such as varying places of latches in the CGRA. So these publications only consider rather homogeneous CGRA designs, in which only the supported instruction classes vary per FU. Some algorithms even seem to rely on the (in our view unrealistic) assumption that all operations have the same latency [Hamzeh et al. 2012, 2013].

In 2013, Kim et al. presented a Fast Modulo Scheduler (FMS) in which the generic NP-hard problem of modulo scheduling for CGRAs becomes tractable by imposing the constraint of following pre-calculated patternized rules [Kim et al. 2013]. As expected, the compilation times are improved by several orders of magnitude, at the cost of code quality (−30% compared to EMS). Through its use of patternized rules, FMS is by construction limited to mostly homogeneous CGRAs. Lee et al. present an integer linear programming approach and a quantum-inspired evolutionary algorithm, both applied after an initial list scheduling [Lee et al. 2011]. Their mapping algorithms adopt high-level synthesis techniques combined with loop unrolling and software pipelining. They also target homogeneous targets.

3. BIMODAL MODULO SCHEDULER

Like EMS, RAMS, and SCCMS, the BMS is a routing-based list scheduler. It differs from the others in four fundamental ways: (i) the supported forms of backtracking, (ii) the priority function that determines in which order nodes are selected to be scheduled, (iii) the (mixed) use of backward and forward routing, and (iv) the used cost functions. The BMS gets its name from the fact that for each attempted II, the scheduler is

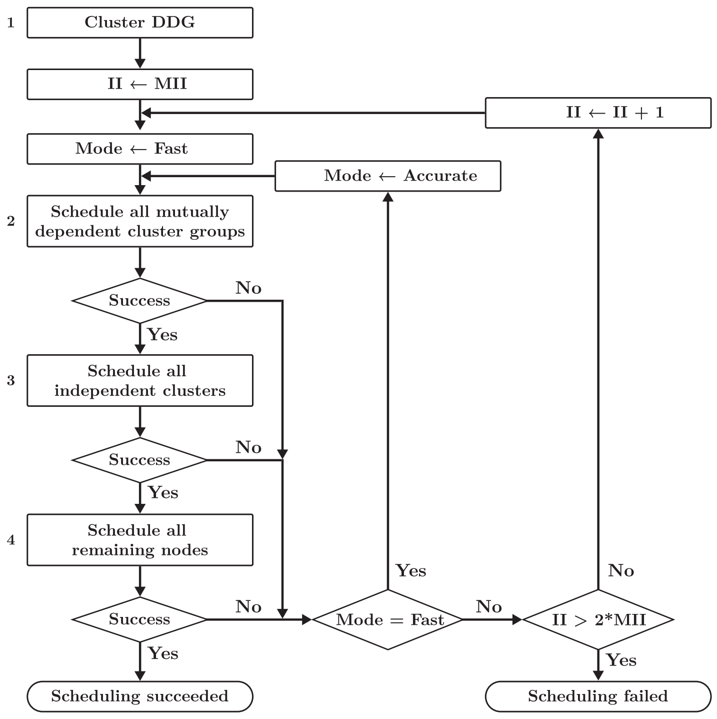


Fig. 4. High-level overview of the BMS scheduling algorithm.

invoked in two modes if necessary. This is shown in Figure 4. The outer loop iterates over increasing IIs. The inner loop can be invoked at most twice, that is, once per mode. In the first, fast mode, simple cost functions are used. For most loops, this mode is successful. If not, then the scheduler is invoked again at the same II in the accurate mode, which is much slower due to its use of more accurate cost functions but which sometimes finds a valid schedule at an II for which the fast mode failed.

3.1. Scheduling Order

3.1.1. Clustering Recurrence Cycles. Recurrence cycles in a DDG result from loop-carried dependencies. The DDG on the right of Figure 2 shows three recurrence cycles corresponding to the three loop-carried dependencies on i, j, and k drawn with dashed arcs. A fourth recurrence cycle (the second for i) is drawn on the top right of the DDG. This recurrence cycle follows from the fact that the loop control is actually converted to `i2 = i + 1; if (i < 512) i = i2 ; else break;`

As mentioned before, the placement of the nodes in recurrence cycles is more constrained than that of other nodes. So as soon as one node in a recurrence cycle is placed, it severely constrains the placement of the other nodes in the cycle. Similarly to what other authors of list schedulers have already proposed, we therefore prioritize the scheduling of nodes in recurrences. We therefore group the nodes into clusters (step 1 in Figure 4). Each cluster corresponds to a recurrence cycle and by scheduling these clusters first (steps 2 and 3). Overall, the strategy is to schedule a cluster as soon as all its predecessors in the DDG (i.e., the cluster's incoming tree) have been scheduled. This also imposes an order on clusters when one of them is part of another one's incoming tree.

Our clustering algorithm creates two kinds of clusters: simple clusters, like clusters 3 and 4 in the DDG of Figure 2, and so-called super clusters, like cluster 1-2 in Figure 2. Super clusters consist of multiple recurrence cycles with a shared header node. The reason for grouping such recurrence cycles into super clusters is that their single common header imposes strict ASAP/ALAP bounds on all of their nodes combined, as they all need to be scheduled within one window of II cycles. In order to maximize the chance of finding valid places for all those nodes, we need to avoid scheduling one cycle completely after the other and instead schedule their nodes together in an order based on priority features such as those nodes' ALAP-ASAP slack. When a node other than a cycle's header node is shared between multiple recurrence cycles, we assign it to one of them instead of grouping their nodes in a super cluster.

3.1.2. Mutually Dependent Clusters. Two or more clusters (be it simple or super clusters) are mutually dependent on each other if there are dependencies from one cluster to the other and vice versa. In Figure 2, clusters 3 and 4 are mutually dependent because of the dependencies through nodes MUL^{11} and MUL^{12}.

Nodes in mutually dependent clusters need to get higher priority than nodes in other clusters: A poor decision for the placement of a node during the scheduling of its containing cluster can make the scheduling of another cluster impossible and thus make the whole scheduling attempt fail. Pro-actively avoiding such failures as first proposed with the EMS scheduler [Park et al. 2008] often does not work well. So a scheduler strategy that aborts a scheduling attempt at some II upon the first failure to place a node will miss many opportunities for generating good schedules. This issue was already observed by the authors of EMS, and as a solution they proposed RAMS. However, RAMS' strategy (i.e., immediately replacing all clusters in a group with mutual dependencies when one of them fails to be scheduled) can also lead to suboptimal results, because it still neglects some placement options.

To avoid (at least part of) those unnecessary failures, we propose *cluster-level backtracking* with Algorithm 1. This algorithm implements node 2 of the flow-chart of Figure 4. It is applied to each group of mutually dependent clusters. Per group, the initialization (line 1) orders the clusters according to the length of their longest recurrence cycle. The center of the algorithm is the call to `ScheduleClusterAndIntraTree` on line 9. This first tries to schedule all nodes in a cluster c_i given an already determined placement of the cluster's header node. Before trying to find a schedule for c_i on line 9, the call to `ScheduleRemainingIncomingTree` on line 6 first schedules all nodes in the incoming tree of c_i that are not dependent on nodes in clusters c_j with $j > i$. This allows `ScheduleClusterAndIntraTree` to take into account the placement of as many as possible nodes in c_i's incoming tree. An invariant at line 9 is that all clusters c_j with $j < i$ have been scheduled, while all clusters c_j with $j > i$ remain to be scheduled later. Inside `ScheduleClusterAndIntraTree`, once a valid placement is found for all nodes in the cluster c_i, the scheduler also tries to schedule additional nodes in the dependence tree between the just scheduled cluster c_i and the other clusters in the group. An attempt to place a node in that intra-tree is made as soon as all of the node's predecessors have been placed.

Whenever the algorithm tries to schedule a cluster c_i and part of the intra-tree of the group, it will attempt this for all valid header placements (in a reasonable window, see the end of this section) by iterating over lines 5–10 for the same i until `ScheduleClusterAndIntraTree` reports success or until all valid placements for the cluster header have been tried. In the case of success, i is incremented on line 12 to let the algorithm start the scheduling of the next cluster c_{i+1} or, if there is no next cluster left, to report overall success (line 20). In the case of failure, that is, when all schedule attempts for c_i at different header placements have failed, the algorithm backtracks

ALGORITHM 1: Scheduling n Mutually Dependent Recurrence Clusters at Some II

```
Input: Group $\mathcal{G}$ of $n$ mutually dependent clusters, value $II$ to attempt schedule
$C = [c_1, \ldots, c_n]$ = clusters sorted in decreasing recurrence cycle length
foreach cluster $c_i$ do
    $TriedHeaderPlacements[c_i] \leftarrow \varnothing$
$i \leftarrow 1$
while $i \le n$ do
    ScheduleRemainingIncomingTree($c_i$,$II$)
    $pos \leftarrow$ NextValidHeaderPosition($c_i$, $TriedHeaderPlacements[c_i]$, $II$)
    if $pos \neq$ NIL then
        $success \leftarrow$ ScheduleClusterAndIntraTree($pos$, $c_i$, $II$)
        $TriedHeaderPlacements[c_i] \leftarrow TriedHeaderPlacements[c_i] \cup \{pos\}$
        if $success$ then
            $i \leftarrow i + 1$
    else
        $TriedHeaderPlacements[c_i] \leftarrow \varnothing$
        if $i > 1$ then
            UnscheduleClusterAndIntraTree($c_{i-1}$, $II$)
            $i \leftarrow i - 1$
        else
            return Failure
return Success
```

on lines 14–17. On line 14, all attempts tried for the current c_i are forgotten to give the greatest possible schedule freedom to potential later scheduling re-attempts for c_i. In case there exists a previously scheduled cluster c_{i-1}, this cluster is unscheduled (line 16), together with the intra-cluster nodes that were scheduled in between c_{i+1} and c_i. Finally, i is decremented to let the while loop continue with the valid next header position for c_{i-1}, if any remain.

For the example of Figure 2, Algorithm 1 is first applied on the group consisting of super cluster 1-2, because that cluster is part of the incoming tree of the clusters 3 and 4 in the other group. Next, the algorithm would be applied on the group consisting of clusters 3 and 4. A trace of that application might look like this:

(1) The incoming tree of cluster 3 is already scheduled, so nothing is done on line 6.
(2) The nodes of cluster 3 are scheduled on line 9, with its header placed in *pos* (i.e., a register x and cycle y in the central RF) as determined on line 7. This means that the value of variable `j` is available in a central register from the cycle y until cycle $y + II - 1$, at which time it is overwritten by the updated value of `j`.
(3) Still on line 9, MUL^{12} is placed by routing the dependency from ADD^{9}.
(4) Iterating over lines 7–10 but never succeeding, the scheduler tries to place cluster 4 at a range of header positions.
(5) All attempts for cluster 4 for the current position of cluster 3 are forgotten on line 14, and cluster 3 and node MUL^{12} are removed from the schedule on line 16.
(6) The nodes of cluster 3 are again scheduled on line 9, with its header placed in a new position pos' determined on line 7.
(7) Node MUL^{12} is placed again on line 9, possibly in another position than in step 3.
(8) The nodes of cluster 4 are now all scheduled on line 9.
(9) Still on line 9, MUL^{11} is placed by routing the dependency from ADD^{10}, after which its other dependencies are routed as well, incl. the one to ADD^{11}.
(10) The algorithm ends successfully.

By means of more backtracking, this algorithm tries more combinations of header positions than EMS and RAMS. This helps in finding better schedules for some loops. Whereas the backtracking may seem exhaustive, we limit its search space in practice: The routine invoked on line 7 only returns cycles in a window of length $2 \cdot II$. Resources are reserved in a modulo reservation table with modulus II, and, except for trivial loops, routing data from any point in the CGRA to any other point can typically be done within II cycles. So when all attempts to place a header in a window of length $2 \cdot II$ and to find a schedule for its cluster have failed, it is typically of no use to look any further.

3.1.3. Multiple Priority Functions, Forward and Backward Routing. Algorithm 1 determines the order in which sets of nodes (incoming trees, intra-trees, and recurrence cycles) are scheduled. It does not determine the order of the nodes within those sets.

Nodes outside recurrence cycles are scheduled in the routines invoked on lines 6 and 9 of the Algorithm 1. Each of those two routines places those nodes in topological order. This is similar to EMS, RAMS, and other list schedulers.

For the nodes in a recurrence cycle, the EMS and RAMS heuristics try to schedule the cycle's nodes in a forward manner. For example, applied to cluster 3 of the DDG in Figure 2, they would first place ADD^9 by routing its dependency starting from the already placed header REG_j. Next, they would place ADD^{11} by routing the dependency starting at the place of ADD^9. Finally, they would try to route the dependency from ADD^{11} back to REG_j, that is, to a write port of the RF in which j is stored.

During our research, we observed that this order often yields suboptimal results. To understand why, it is important to consider that, unlike in Figure 2, many loops feature header nodes with many outgoing dependencies, of which only one or few are typically on the critical path of the recurrence cycle. When a header node is placed on a register in the central RF in some cycle y (on line 9 of Algorithm 1), the scheduler essentially decides that the value of the register will remain constant (and hence available for all outgoing dependencies of the header) from cycle y to cycle $y + II - 1$. While the outgoing dependencies of a header can be routed starting at that register's RRG node in any of the cycles from y to $y + II - 1$, the header's incoming edge needs to be routed to arrive exactly in cycle $y + II - 1$.

For that reason, and in addition to most architectures featuring fewer write ports than read ports to the central RF, the header's incoming edge is much more constrained. Routing that edge first improves the chance of finding a valid schedule, in particular for lengthy recurrence cycles. The BMS therefore places the nodes in recurrence cycles in a reverse order, that is, in the opposite order of the one presented for EMS and reused for RAMS. In doing so, the BMS also routes the dependencies backwards. For cluster 3 of our example DDG in Figure 2, the invocation of `ScheduleClusterAndIntraTree` on line 9 of Algorithm 1 will first place ADD^{11} by finding the cheapest backwards route starting from a central RF write port selected when the header REG_j was placed. Next, it will place ADD^9 by finding the cheapest backwards route starting from the input ports of the FU on which ADD^{11} was placed. Finally, the edge from REG_j to ADD^9 will be routed.

3.1.4. Placing and Routing Constant Nodes. So far, we have not discussed when or how the constant operand nodes are placed onto CMs. To the best of our knowledge, other articles presenting list schedulers do not discuss the placement of constants at all, that is, neither the hardware approach to store constants nor the supporting compiler techniques are discussed. For our designs with CMs, we observed that unless a CGRA is bloated with wide CMs all over the array, the constant operands found in many DSP kernels, such as Fast Fourier Transform (FFT) butterfly operands and Finite

Impulse Response (FIR) filter tap values, can easily become the critical nodes that the scheduler needs to get absolutely right. Moreover, as such loops typically contain many small constants (used for, among others, memory indexing), as well as many large values (the butterfly operands and the tap values), significant power and area reductions can be obtained by not bloating the architecture with wide CMs but by using heterogeneous, scarcer ones instead, as shown in the example of Figure 1(b).

To get the allocation of constants to CMs right, the BMS deploys four new heuristics:

(1) Each constant operand gets a separate node in the DDG. This allows the scheduler to place all of them independently, even if the operands of multiple instructions happen to have the same value.
(2) Constant memories can be overused by multiple constant operands with the same value, and so can all resources used in nets from constant operands to their operations. This allows the scheduler to reuse resources for multiple identical constant operands when this is beneficial.
(3) Constant operands are placed right after their operation is placed by finding a backward route from that operation's FU to a constant memory. This ensures that as few as possible resources are used to route constant operands to their operations.
(4) In fact, the placement of the constant operand of an operation during the scheduling of clusters and trees on lines 6 and 9 is treated as if it is an integral part of the placement of the operation itself. This shows in two ways. First, as Section 3.2.1 will discuss, the availability of close-by and wide-enough constant memories is taken into account when trying to find a place for the operation. For example, in the example DDG of Figure 2, the availability of a close-by CMs for constant value 2 becomes part of the cost function used to find the cheapest backward route from the place of ADD^{11} to potential places for ADD^{9}. Second, if, after the successful placement of the operation the placement of the constant operation fails, then the algorithm backtracks and searches for another place for the operation rather than immediately returning failure for the incoming tree or for the cluster at one of its header's positions.

3.2. Bimodal Cost Functions

Before a DDG node is placed on the MRRG, the scheduler computes so-called routing costs for all the MRRG nodes. In Algorithm 1, this computation is embedded in the routines invoked on lines 6, 7, and 9. As first proposed in EMS, these routing costs are then used to find the best place by finding the cheapest route to a suitable place. The routing cost functions are designed with two goals in mind. First, the goal is to minimize the number of hardware resources used for each placed/routed DDG node/edge. This naturally keeps resources available for the P&R of later nodes and edges. Second, we try to avoid future P&R failures proactively by penalizing the allocation of resources that might block future routes and by trying to reserve scarce resources (in heterogeneous CGRAs) for *expensive* operations, that is, operations that need scarce resources.

Not surprisingly, the routing cost functions in the BMS are inspired by those in EMS. We do propose some significant changes, however, in several directions. Most importantly, we simplify the most complex, computation-intensive cost function term to reduce compilation times. On top of that, we propose two alternative modes for that cost function term computation, of which one is orders of magnitude faster. Last but not least, we compensate the loss in quality following from that simplification by adapting the other cost function terms without significantly increasing their complexity.

3.2.1. Goal 1: Minimizing the Used Resources. To guide the scheduler into minimizing the number of resources allocated during each P&R of a node, EMS' authors proposed to use two cost terms per resource in their resource reservation table. We use the same cost terms but assign them to nodes in the MRRG instead.

The first cost is a *static* cost C_{static}. Each node gets a positive static cost, such that, all other things being equal, the P&R will favour routes that occupy fewer resources.

The second cost is a so-called *affinity* cost C_{aff}, which EMS actually reused from MGE. When the scheduler tries to find an FU in the MRRG to place some DDG node a, of which neighbouring operations in the DDG have already been placed, this cost term makes FUs in the neighbourhood of the already placed ones cheaper.

The affinity value of a pair of operations (a, a') in a DDG reflects their proximity in the DDG and is calculated with Equation (1). $num\_cons(a, a', d)$ denotes the number of common consumers of a and a' at distance d in the DDG, at a maximum distance of max_dist. If we denote the set of already placed nodes with A, the MRRG node occupied by the placed node a' as $place(a')$, and the distance in hops between two MRRG nodes with $dist$, then the affinity cost $C_{aff}(a, n)$ of an MRRG FU node n for the placement of a is as shown in Equation (2):

$$affinity(a, a') = \sum_{d=1}^{max\_dist} 2^{max\_dist-d} \cdot num\_cons(a, a', d), \tag{1}$$

$$C_{aff}(n, a) = \sum_{a' \in A: affinity(a,a')>0} \frac{distance(n, place(a'))}{affinity(a, a')}. \tag{2}$$

Consider the example of Figure 2 again, and the second-to-last step of the compilation, when only MUL^{11} remains to be placed. To find a good place, the router will try to find the cheapest route from the FU node occupied by ADD^{10} to an FU node that can execute a MUL. Since MUL^{11} has high affinity with ADD^{11}, MRRG nodes in the immediate neighbourhood of the FU node occupied by ADD^{11} will get a low affinity cost, while nodes far away will get a higher affinity cost. This way, the cheapest path that the router is trying to find to route the dependency from ADD^{10} to MUL^{11} will already be steered into the direction of ADD^{11}, in an effort to reduce the future resource consumption when the dependency from MUL^{11} to ADD^{11} will be routed later.

As already stated, the authors of EMS and RAMS pay no attention to constant operands and possible heterogeneity of CMs. The discussed affinity cost, reused directly from the EMS, does nothing towards finding good allocations for constant operands. Given that the BMS places constant operand nodes after their operations, the discussed affinity cost cannot avoid placements far away from needed CMs, and hence it can easily lead to problematic schedule decisions, as we observed for DSP kernels with many different constants, such as FFTs, Discrete Cosine Transforms (DCTs), and FIR filters. Going back to the example of Figure 2, the problem is that when an operation with a constant operand is placed on the top right FU, the constant operand stored in one of the memories can only be routed to that top right FU through one of the bottom FUs. In that case, one of those bottom FUs will be occupied for one cycle simply to route data, thus preventing the FU from performing useful computations instead.

To prevent such bad placements as much as possible, we propose to complement the existing affinity cost with a so-called *constant affinity cost*C_{const_aff}. For routing dependencies to operations with a constant operand (FU, cycle), slots in the MRRG get a constant affinity cost that is inversely proportional to the availability of suitable (CM, cycle) slots in their neighbourhood. The “neighbourhood” of an FU in this context is defined as all CMs that are connected directly to the FU, that is, without needing to go through another FU. A CM is suitable if it is wide enough to store the constant. A CM is considered available in cycle y_c for some FU in cycle y_o if the path from the CM to the FU takes exactly $y_o - y_c$ cycles, and if the CM is (1) either not yet occupied in cycle y_c or (2) if it is already occupied with the same constant. Furthermore, the constant affinity cost depends on the scarcity of unoccupied CM slots and on the future demand for them: When more and more different constant operands still remain to be placed in the future, and when fewer CM slots remain available, the constant affinity cost associated with unoccupied CM slots becomes higher, while the cost associated with CMs already occupied with the same constant value becomes lower. This way, the scheduler proactively tries to aim for resource sharing for constant operands when useful, but it does not enforce resource sharing when its not necessary.

Equation (3) defines $C_{const\_aff}(n, a, c)$, the cost term for the FU node n in the MRRG when trying to place an operation a from the DDG with a constant operand with value c. c takes the value NIL for an operation with no constant operand. $NCM(n)$ denotes the set of neighbouring CMs of FU MRRG node n. $R_M(c)$ represents the number of available CMs that are wide enough to store c. If, while trying to place any node on the MRRG, $R_M(c)$ becomes zero for any as-yet unplaced c, then the scheduler considers that placement attempt a failure, and it will abort its scheduling attempt or backtrack. So whenever the $C_{const\_aff}(n, a, c)$ needs to be computed for an as-yet unplaced c, $R_M(c)$ is strictly positive. C_d is the number of constant nodes with distinct values still to be placed, and C_a is the total number of all constant nodes still to be placed. The factor2 in the third case of the equation was determined experimentally. $C_d + 2C_a$ models the constant memory pressure; the factor 2 proved to give the best results:

$$C_{const\_aff}(n, a, c) = \begin{cases} 0 & \text{if } c \neq NIL \wedge \ \exists m \in NCM(n) : m \text{ already stores } c \\ \infty & \text{if } c \neq NIL \wedge \ !\exists m \in NCM(n) : m \text{ is available for } c \\ \frac{C_d + 2C_a}{R_M(c)} & \text{otherwise} \end{cases}. \tag{3}$$

3.2.2. Goal 2: Reserving Critical Resources. Similarly to EMS, the BMS tries to reserve critical MRRG nodes by making them more expensive when the probability of needing them for future routes increases. A completely accurate calculation of this probability and its mapping to a *probability cost* is complex to calculate. So, instead, we aim for an approximation. We observed that the approximation in EMS is very accurate but also time-consuming. For that reason, we propose to adapt the EMS cost functions and to extend them. In line with EMS, we consider two forms of probability costs.

Expensive Resource Costs. This first form relates to heterogeneity. When some resource is scarcer than other resources, and when the demand for that resource is high, it needs to be reserved for operations that really need that resource. So for any FU that supports more types of operations than other FUs, such as FUs that can execute LD/ST or MUL operations on top of standard ALU operations, we consider an *expensive* resource cost as expressed by Equation (4). $EOP(n)$ denotes the set of all expensive instructions in the instruction set supported by the FU n. Furthermore, $O(i)$ denotes the number of as-yet unplaced DDG operations of type i, and $R_F(i)$ denotes the number of FU slots that are still available in the modulo schedule for executing that type of operation. These cost function terms are borrowed directly from EMS.

$$C_{exp}(n) = \sum_{i \in EOP(n)} \frac{O(i)}{R_F(i)}. \tag{4}$$

On top of these terms, however, the BMS also considers expensive resource costs that do not reflect the scarcity of appropriate FUs per se but instead reflect their role in routing dependencies to and from the central RF. Many loops feature many live-in variables, such as multiple base addresses and induction variables. As those variables are placed in the central RF, and both the connections to the central RF and the

FUs directly connected to the central RF become scarce resources simply because of their position in the CGRA. So we propose to add two more *expensive routing resource costs*.

The first one is $C_{exp\_route\_P}(n) = O_P/R_P$. This cost is computed for MMRG nodes that model read and write ports to the central RF. For other nodes it is set to zero. For such read/write port nodes, O_P is the number of outgoing/incoming edges of register nodes in the DDG still to be routed, and R_P is the number of read/write ports still available in the schedule. With this cost, the scheduler avoids unnecessary uses of these ports when they are under high demand. The second cost function is $C_{exp\_route\_F}(n) = O_F/R_F$. This cost function is only computed for FU nodes in the MRRG that are connected directly to the central RF's read and write ports. O_F is the number of as-yet unplaced FU operations that are directly connected to register nodes in the DDG. R_F is the number of slots in the schedule in which FUs directly connected to the central RF are still available. This cost function helps the scheduler reserve FUs close to the central RF for operations that can benefit from being placed there.

Probability Routing Costs. The second form of probability costs relates to already placed nodes from the DDG. When the tail or head node of a dependency is already placed, but the dependency is not routed yet because the other node is not placed yet, we know where the routing of that dependency will start. In the remainder of this section, we refer to such dependencies as half-placed dependencies.

Knowing the possible places where the target operation of the half-placed dependencies might still be placed in the future, we can compute which resources will be more likely needed for routing the dependencies and which ones will not be needed (likely). To try to avoid that resources are consumed that need to be reserved for later routing, we hence at all times assign a probability routing cost to all MRRG nodes.

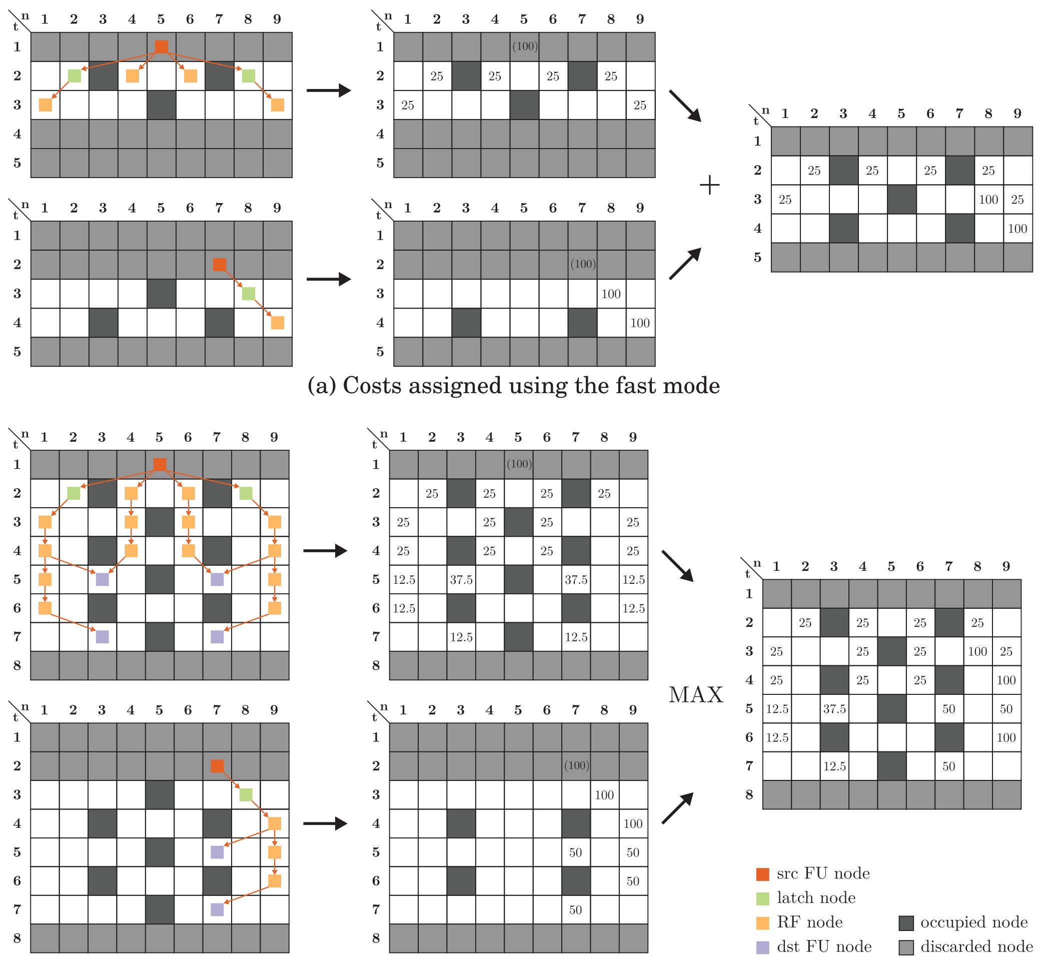
As the probability that some resource will be needed for later routing depends on the already-placed nodes, the associated cost needs to be recomputed after every P&R decision. As proposed by the authors of EMS, the probabilities can be computed accurately by pre-computing all possible routes for all half-placed dependencies to all possible places for their nodes still to be placed and by deriving, for each node in the MRRG, the likelihood that at least one route will occupy the node. Such an accurate estimation is time-consuming, however, and it needs to be executed very frequently.

For this reason we propose to replace the EMS probability cost function by two novel ones, corresponding to two modes that require much less computation time. These are used in the fast resp. accurate modes that were introduced before.

The Fast Mode. Per half-placed dependency, this mode traverses the MRRG starting from the placed node until entry nodes to either a suitable FU (for the operation still to be placed) or any RF are reached within a static, narrow search window. The traversal moves forward/backward when the producing/consuming node of the half-placed dependency was already placed.

The probability cost is then evenly divided over all the discovered paths. Per MRRG node, all probability costs thus computed for all half-placed dependencies are then simply accumulated. Computing the costs per half-open dependency requires only two straightforward passes over the nodes on the discovered paths in the MRRG, which are always very short because of the short search window.

The core idea behind this very simple cost function is the assumption that whenever a route can be found to some RF, in which values can stay anywhere between 0 and $II - 1$ cycles, it will highly likely be possible to route from that RF to a suitable FU along many possible paths. So the only routes to consider as more likely needed in the future are those leading directly to a suitable FU and those leading to RFs.



(a) Costs assigned using the fast mode

(b) Costs assigned using the accurate mode

Fig. 5. Assigning probability costs in the BMS's two modes.

The algorithms to compute these cost functions are simple enough to be explained by means of an example. For the sake of clarity, the illustration in Figure 5(a) depicts a simplified view on the MRRG in the form of a resource reservation table. On the left, the possible forward paths from two already-placed producer nodes (on FU resources 5 and 7) to some RFs (resources 1, 4, 6, and 9) are enumerated. In the center, costs are assigned to each node on a found route. For each already-placed node, there is a starting cost of 100. Whenever a route diverges into multiple ones, the assigned costs are simply distributed equally. On the right, the costs for the two half-open dependencies are aggregated by means of summation. The aggregated costs, for all yet-to-be-routed half-open dependencies form the final costs $C_{prob\_fast}(n)$. In this example the search depth was set to 2 cycles. In practice, the depth is set to 3 to 5 cycles, depending on how many cycles it takes in the architecture to route a value from one FU to another, that is, how deeply pipelined the connections in the CGRA are.

The Accurate Mode. This mode simulates real routing attempts for half-placed dependencies. In this mode, the search does not stop at entry nodes to RFs but instead

Table I. Comparison of the *Fast* and *Accurate* Modes of the Bimodal Probability Cost Function

Mode	**Search Window**	**Forward Depth**	**Backward Depth**
Fast	Static	static value (3–5)	static value (3–5)
Accurate	Dynamic	$ASAP_{cons} + II - t_{prod}$	$ASAP_{prod} - t_{cons}$

continues until a suitable FU is found or the boundary of the search window is reached. The depth of the window in this mode is dynamic and linked to the ASAP time of the unplaced node as shown in Table I, where t_{prod} and t_{cons} denote the times at which the producer (consumer nodes) are already placed, and $ASAP_{cons}$ and $ASAP_{prod}$ denote the ASAP times of the nodes still to be placed.

The accurate mode thus explores different routes for each half-placed dependency, but the cost distribution among diverging routes is the same as in the fast mode. When multiple paths converge in some slot, the costs of those paths are simply added up (as is done, by the way, in the fast mode). This is demonstrated in the top part of Figure 5(a) where, for example, resource 3 in cycle 5 gets a cost of $37.5 = 25 + 12.5$. After repeating the cost computation for all half-placed dependencies, the costs are now aggregated into costs $C_{prob\_acc}(n)$ with a MAX function instead of by summation[1]. Figure 5(b) illustrates the accurate mode for an II of 2.

3.2.3. Total Cost Function. Per MRRG node, the total cost function consists of a weighted sum of all discussed cost functions. In this regard, the BMS is identical to EMS and RAMS.

4. EVALUATION

4.1. Methodology and Experimental Setup

As a measure of code quality, we will use MII/II, the fraction of the theoretically optimal performance that a scheduler achieves. Higher is better; MII/II=1 for optimal code. We use this measure for individual loops and for whole programs. In the latter case, MII and II denote the accumulated MIIs and IIs over all a program's loops.

In a world where neither the benchmark applications nor the schedulers or CGRA designs are shared, performing fair comparisons with existing techniques is challenging. Besides the BMS, we only have access to DRESC, so we will only report measured results for the BMS and DRESC. Any comparison to DRESC poses a problem, however, because of DRESC's dependence on PRNG seeds, as explained in Section 2.1. To illustrate the problem, consider the following experiment for any positive integer value n: First, randomly pick n PRNG seeds. Then compile one loop after the other. For each loop, try one seed after the other. As soon as a schedule is generated at the loop's MII, move on to the next loop, skipping seeds if any remain. For each loop, select the lowest achieved II. As a metric for code quality, sum up the IIs selected for all loops. As a metric for compilation time, sum up all compilation times of all executed scheduling attempts.

Figure 6(a) shows the outcomes of 100 experiments with 100 different random seeds and with $n = 1$ for our MPEG benchmark. Both the code quality and the compilation time vary significantly depending on the seed. Figure 6(b) adds the outcomes of 100 experiments with $n = 2$. Figure 6(c) adds those for $n = 3$ and $n = 4$. Clearly, compilation time and code quality can be traded off with DRESC by adopting different strategies. The results of any experiment therefore depend on the amount of "training" that was

[1]While the cost values in the example reflect probability percentages to ease human understanding, they are costs and not probabilities, so the aggregation does not need to be consistent with probability theory.

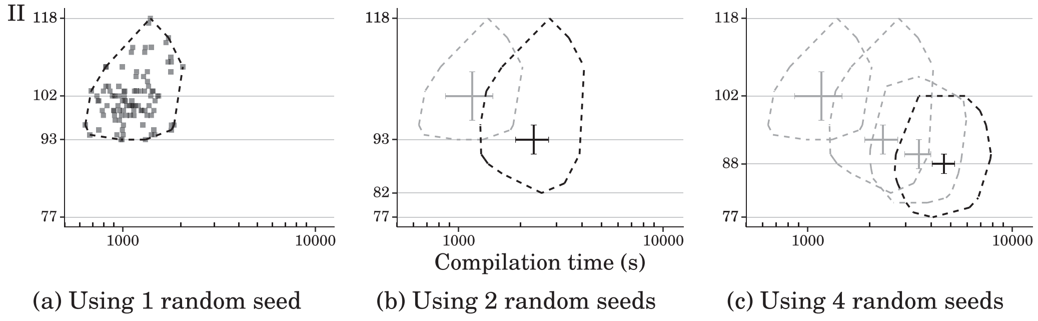


Fig. 6. DRESC performance on the MPEG benchmark.

conducted to find good random seeds and on which parts of the compilation strategy and training are included in the reported compilation times.

No authors ever discussed this aspect, however. Past authors only stated that their scheduler is Δx faster than DRESC or than their own (partial) re-implementation of DRESC and that it produces Δy better or worse code for some set of benchmarks. It is not at all clear how to interpret their Δx and Δy, however, because it is unclear which point(s) within all of the DRESC hulls they used as the baseline. In fact, they might have used a completely different compilation strategy of which the compilation time and achieved results lay far outside the hulls we measured.

To compare the BMS and DRESC more transparently, we report hulls for DRESC. For BMS compilation times, we always present the average times of 10 compilations. We measured the clock-wall compilation time and achieved IIs for DRESC and the BMS on a single core of an otherwise unloaded Intel Core i7-5930K CPU clocked at 4.2GHz with 15MB L3 Cache and 16GB DDR4 RAM clocked at 2.4GHz.

In addition, we will report compilation times and MII/II results for an oracle version of DRESC, which we will denote as DRESC*. This oracle version is DRESC configured to deploy the best possible compilation strategy: It is invoked with the best possible random seeds as obtained from an oracle instead of through time-consuming training experiments. As such, the results reported for DRESC* present an upper bound on the compilation speeds and code quality that can be achieved with DRESC-like compilers.

We evaluate the BMS and DRESC on three benchmarks that were previously developed at IMEC while doing software-hardware co-design space exploration in the domains of Multimedia (MM) processing and SDRs. From the MM domain, we map an MPEG decoder and an H.264 video decoder, totalling 53 loops to be compiled for the ADRES' CGRA mode, onto a heterogeneous ADRES instance designed for that domain. By means of manually tuned loops (including many intrinsics), the benchmarks exploit the heterogeneous ADRES instance that includes 4×8-bit SIMD and domain-specific instructions, such as clipping operations. Both the benchmarks and the ADRES instance have already been presented and used in the literature to evaluate DRESC [Mei et al. 2008; De Sutter et al. 2009]. For the SDR domain, we map a collection of 19 manually tuned DSP kernels onto a 4×4 CGRA optimized for the SDR domain. This is a heterogeneous design that supports 4×16-bit SIMD and several domain-specific instructions to perform fixed-point complex integer computations. These kernels and this ADRES instance have also already been presented and used in the literature [Bougard et al. 2008; Novo et al. 2009; Derudder et al. 2009].

The targeted ADRES instances are heterogeneous in several ways. They feature a central RF that is connected directly to the top CGRA row but indirectly to the lower rows. Even on the top row, not all FUs have direct access to the central RF. The number

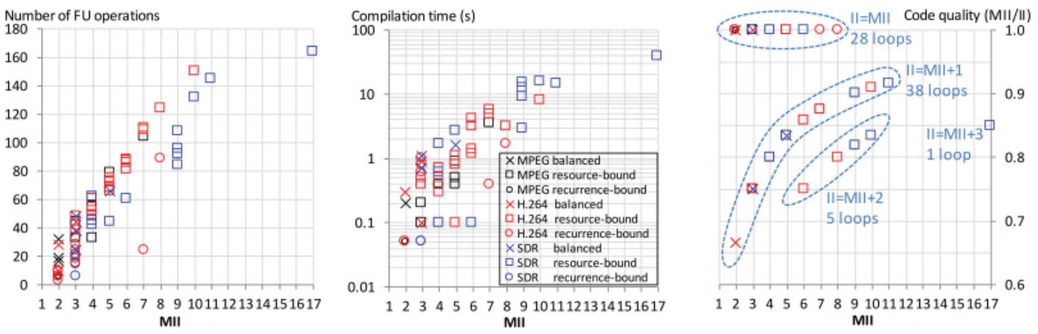


Fig. 7. Overview of loop sizes, BMS compilation time, and code quality.

of ports to that RF is limited to lower its power consumption, so only three of the four FUs of the top row are connected directly to input/output ports of the RF. The CGRA instances feature a limited number of load/store units dispersed throughout the array. Similarly, they feature a limited number of multiplication units dispersed throughout the array. Furthermore, some domain-specific instructions have three source operands instead of two. Those instructions can only be executed on the FUs supporting stores, as those FUs already feature three source operands ports to accommodate a base address, an offset, and the actual data of store operations. Moreover, the CGRAs feature constant memories of variable, reduced word widths to save on power consumption, and the interconnect between the FUs and the RFs is heterogeneous.

From each of these two existing CGRA instances, which we will denote as variant 0, we derived three other variations on which we also evaluated the BMS and DRESC. Variant 1 also incorporates the dedicated RFs proposed by Oh et al. in the article in which they compared RAMS and HBEMS to DRESC [Oh et al. 2009]. Variant 2 is a more homogeneous design, in which all constant memories are 32-bits wide, in which the central RF features enough ports for direct connection to all FUs on the top row of the CGRA, and in which the interconnect is a homogeneous mesh interconnect. Variant 3 is the more homogeneous design of variant 2 plus the dedicated RFs of Oh et al.

4.2. Experimental Results and Comparison with Different Schedulers

The left chart in Figure 7 visualizes the number of FU operations in each loop's DDG, each loop's MII on the variant 1 CGRA design, and whether each loop is resource-bound (ResMII>RecMII), recurrence-bound (ResMII<RecMII), or balanced (ResMII=RecMII) on those CGRAs. For our three benchmarks, the average loop MIIs are 3.76, 4.28, and 6.31. For the multimedia benchmarks, the minimal MII is 2, and for the SDR benchmarks it is 3. This is due to the design of our CGRAs' interconnects as shown in Figure 3. The latches inserted in those interconnects effectively pipeline the CGRAs' data paths. They do not impact the IPC significantly because the scheduler uses them as temporary storage and because most loops remain resource bound, despite their increased RecMII. The latches have a huge effect on achievable clock speed, however.

The middle and right charts in Figure 7 display the BMS's compilation times and code quality. The compilation times scale exponentially with the loop sizes, and the vast majority of the loops get scheduled at their MII or MII+1. Both scaling trends are in line with the other schedulers against which we compare the BMS in this evaluation.

Figure 8 reports the overall code quality and compilation times measured for the BMS, DRESC, and DRESC* for the benchmarks compiled for variant 1. Figure 9 reports the code quality and compilation times measured for the BMS and DRESC* for all four design variants and for the MPEG (M), H.264 (H), and SDR (S) benchmarks.

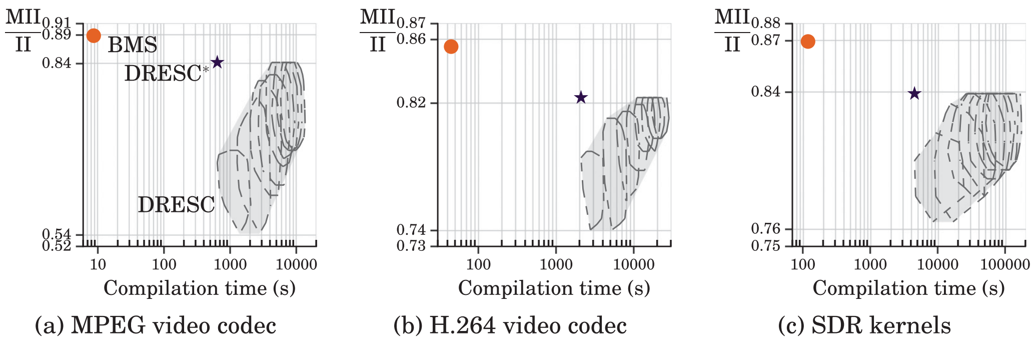


Fig. 8. Results of BMS, DRESC, and DRESC* when compiling for variant 1.

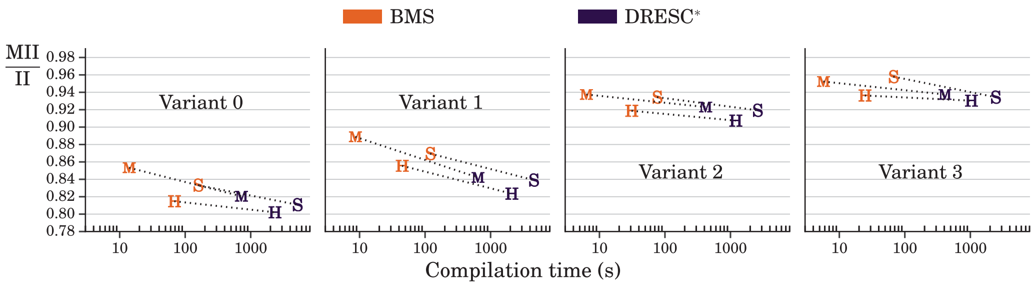


Fig. 9. MPEG, H.264, and SDR results with the BMS (orange) and DRESC* (blue). Corresponding results are connected to highlight the improvement that the BMS brings over DRESC*.

First, it is clear that the more homogeneous CGRAs are much easier targets: Compilation is much faster, and much higher MII/II values are achieved for variants 2 and 3, with both the BMS and DRESC*. Second, it is clear that the BMS outperforms DRESC* both in terms of code quality and compilation speed on all four variants of the two CGRA designs. Third, the BMS code quality improvements over DRESC* are much more pronounced for the more heterogeneous CGRA designs. On variant 1, the BMS produces MII/IIs between 0.030 and 0.047 higher than DRESC*. For variant 3, the MII/IIs of the BMS are only between 0.006 and 0.024 higher than those of DRESC*. For variant 1, the BMS produces code between 38× and 74× faster than DRESC*. For variant 3, the speed-up varies between factors 36× and 72x×, averaging at 40×.

To compare the BMS to RAMS and HBEMS, the bottom half of Table II incorporates all experimental results available from Oh et al. [2009]. The top half of the table presents the main BMS results in a similar structure to facilitate comparison.

The table combines results for five different CGRA designs. CGRA variants 4 and 5 in this table are the 4 × 4 CGRA instances targeted by Oh et al., with and without the dedicated RFs that have been mentioned before. While not all details about their designs are publicly available, in terms of homogeneity, variant 4 most closely resembles variant 3, and variant 5 most closely resembles variant 2.

The table also combines results for 9 different benchmarks. In terms of complexity, measured in the form of average MII per loop, the 6 benchmarks used by Oh et al. cover about the same range as the 3 benchmarks we compiled with the BMS. Moreover, we know that Oh et al. manually tuned their loops, as was done for our loops.

Compilation times are included for three host CPUs, as indicated next to the techniques in the three rightmost columns. CPU 1 is the 4.2GHz Intel Core i7-5930K with

Table II. BMS Results (Top Half) and RAMS/HBEMS Results from Oh et al. [2009] (Bottom Half)

CGRA variant	benchmark	features			code quality (∑ MII / ∑ II)			compilation time (∑t / ∑ MII in seconds)		
		#loops	∑ MII	average MII	DRESC*		BMS	DRESC*-1	BMS-1	BMS-2
0	MPEG	17	64	3.76	0.82		0.85	11.23	0.22	0.49
	H.264 (*)	36	154	4.28	0.80		0.81	15.14	0.44	1.04
	SDR	19	120	6.32	0.81		0.83	42.85	1.31	2.95
1	MPEG	17	64	3.76	0.84		0.89	9.98	0.13	0.27
	H.264 (*)	36	154	4.28	0.82		0.86	13.60	0.29	0.54
	SDR	19	120	6.32	0.84		0.87	37.83	1.00	2.22
2	MPEG	17	60	3.53	0.92		0.94	6.93	0.11	0.18
	H.264 (*)	36	147	4.08	0.91		0.92	8.14	0.21	0.46
	SDR	19	114	6.00	0.92		0.93	22.62	0.66	1.45
3	MPEG	17	60	3.53	0.94		0.95	6.80	0.10	0.18
	H.264 (*)	36	147	4.08	0.93		0.94	7.01	0.17	0.35
	SDR	19	114	6.00	0.93		0.96	21.25	0.59	1.32
					DRESC	HBEMS	RAMS	DRESC-3	HBEMS-3	RAMS-3
4	3D rendering	80	290	3.63	0.95	0.87	0.92	151.16	0.77	0.55
	AAC dec	35	123	3.51	0.87	0.71	0.90	282.84	0.13	0.24
	AMR-WB+ dec	44	230	5.23	0.99	0.93	0.98	87.89	0.14	0.35
	eAAC+ dec	47	204	4.34	0.94	0.82	0.95	123.97	0.12	0.31
	H.264 dec (*)	61	387	6.34	0.82	0.84	0.85	146.78	1.19	1.75
	mp3 dec	8	44	5.50	0.98	0.81	0.98	393.49	0.29	0.69
	MPEG surr. dec	115	522	4.54	0.94	0.90	0.94	125.40	0.10	0.98
5	3D rendering	80	290	3.63			0.84	(*): We do not know to what extent the source code and the selected loops of the two evaluated versions of the H.264 benchmarks (one for BMS, one for RAMS/HBEMS) overlap or differ.		
	AAC dec	35	123	3.51			0.77			
	AMR-WB+ dec	44	230	5.23			0.94			
	eAAC+ dec	47	204	4.34			0.82			
	H.264 dec (*)	61	387	6.34			0.71			
	mp3 dec	8	44	5.50			0.71			
	MPEG surr. dec	115	522	4.54			0.80			

15MB of L3 cache on which we ran our main experiments. We re-measured the BMS's compilation time on an older CPU 2, a 3.0GHz Intel Core 2 Duo E8400 with 6MB L2 cache, to bridge part of the performance gap between our state-of-the-art CPU 1 and the older CPU 3 used by Oh et al., about which the only public information is that it concerns a 2.66GHz Intel Dual-Core Xeon. (The total memory size is mostly irrelevant, we have never seen a CGRA compiler require more than 30MB to compile a loop.)

Oh et al. essentially used the same DRESC we used. As already explained in Section 4.1, DRESC is the same compiler as DRESC*, the only difference being that DRESC* denotes the results of invoking DRESC with well-chosen parameters (incl. random seeds) rather than with a range of parameters that the compiler should explore. As they confirmed to us, the very long compilation times Oh et al. reported for DRESC-3 in Table II indicate that they also included a significant, but not precisely described, amount of parameter and random seed exploration in their DRESC measurements. It is therefore no surprise that the code quality results reported for DRESC* and DRESC on variants 3 and 4, respectively, are similar.

Comparing the results of the more homogeneous CGRA variants 3 and 4, which both feature the dedicated RFs, it is clear that the BMS is comparable to RAMS in terms of code quality and compilation speed. With the BMS, we did not observe a code quality drop when omitting those RFs; however, the drop from variant 3 to 2 and from variant 1 to 0 with the BMS is much smaller than the drop reported by Oh et al. from variant 4 to 5. This is mostly due to the BMS's use of the novel expensive routing resource costs, as introduced in Section 3.2.2. Moreover, the BMS is able to improve the results obtained with DRESC* slightly but also consistently for all variants 0 to 3. By contrast, for variant 4, RAMS was not able to consistently improve the results reported for DRESC.

Figure 10 shows the results of the only GraphMinor experiment for which code qualities and compilation times are reported for the 18 loops evaluated in Chen and Mitra [2014]. This experiment targeted a homogeneous 4×4 CGRA without a central RF, without pipelining latches between the FUs (as implied by some loops having $MII = 1$), and in which all FUs can execute all operations, incl. memory accesses.

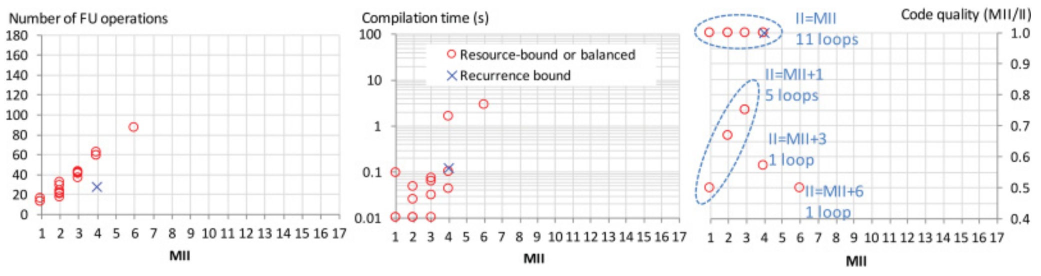


Fig. 10. GraphMinor results for comparison with the BMS results of Figure 7.

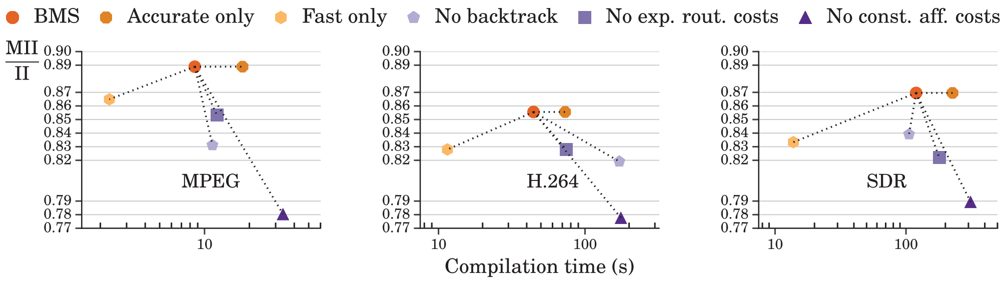


Fig. 11. Impact of the BMS's unique features on scheduling performance.

GraphMinor's compilation times were measured on a 2.83GHz Intel Quad-Core with 12MB L3 cache.

Several observations can be made when comparing Figures 10 and 7. First, GraphMinor was evaluated on significantly fewer and simpler loops than the BMS. This makes it hard to draw conclusions regarding the compilation speed of the BMS and GraphMinor. Given the differences in host CPUs and in target architectures, it is not possible to announce a clear winner. For code quality, however, there is some indication that the BMS outperforms GraphMinor for more complex loops. For one loop, GraphMinor only achieves an $II = MII + 6$. For a range of other 4×4 architectures, GraphMinor got the difference between MII and II down to 4 for that loop but never below it. For another loop, GraphMinor never gets the difference down below 3. By contrast, on variants 2 and 3, which are similar to GraphMinor's targets in terms of heterogeneity, the BMS always finds schedules with an $II \leq MII + 2$. In fact, for only one loop (the one with $MII = 17$ in Figure 7), the BMS needed two more cycles on top of MII. For all the other loops scheduled onto variants 2 and 3 with the BMS, $II = MII$ or $II = MII + 1$.

In REGIMap's evaluation, neither the number of compiled loops per benchmark nor their complexities are mentioned [Hamzeh et al. 2013]. So comparing REGIMap and BMS compilation times is impossible. MII/II values are presented per benchmark, but the article does not mention whether those are computed as $\sum MII / \sum II$, or as $avg(MII/II)$. Moreover, only fully homogeneous CGRAs are targeted, with varying amounts of local registers. For their easiest compilation target (with the most local registers), they report MII/II values in the range [0.69,1]. This overlaps with the range obtained with the BMS.

4.3. Impact of the Different BMS Contributions

To assess the impact of the different contributions presented in Section 3, Figure 11 visualizes the effect on the compilation times and the achieved code quality when we

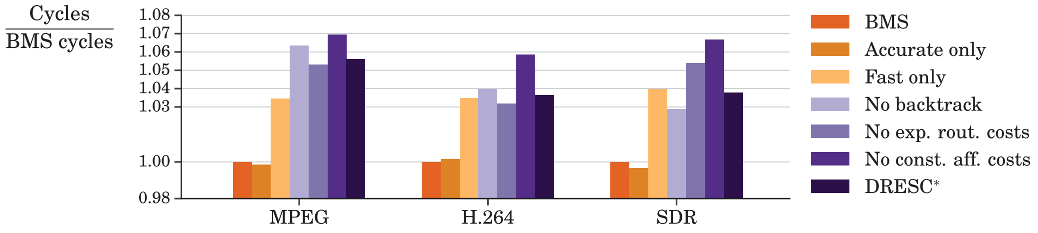


Fig. 12. Impact of the BMS's unique features on whole benchmarks' CGRA execution time.

individually disable or omit those contributions from the BMS while targeting ADRES instances of variant 1. These contributions are the fast mode and the accurate mode of Section 3.2.2, the back-tracking scheme of Section 3.1, the expensive routing resource costs of Section 3.2.2, and the constant memory affinity costs of Section 3.2.1.

Omitting the fast mode only has an impact on compilation times. This is not surprising, as the fast mode's only goal is to try to find the same II as the accurate mode, but to do so much faster. The fast mode spends much less time computing probability costs, and when it fails for some tried II, it fails much earlier, after fewer DDG nodes have been placed. Compared to the accurate mode, the fast mode's compilation time on any loop, at any tried II is therefore almost negligible. So in the BMS, little compilation time can be saved by omitting the fast mode. By contrast, omitting that fast mode implies that the accurate mode needs to be invoked one additional time for every loop for which the fast mode in the BMS found a schedule first. In our experiments, the fast mode succeeded to do that for 58 of 72 loops. The net result is that the compilation time increases significantly by omitting the fast mode. When the accurate mode is omitted, compilation becomes much faster, but code quality drops as well. The reason is, of course, that the fast scheduler sometimes (i.e., 14/72 times) fails to find schedules at the same low IIs found by the accurate mode. In summary, we can conclude that both the fast mode and the accurate mode contribute significantly to the BMS.

For the other evaluated contributions of Section 3, we observe a significant decrease in code quality when they are omitted. So all of them also contribute significantly to the BMS on our target CGRAs. In almost all cases, we also observe an increase in compilation time when the contributions are omitted. The reason is that they require very little computation time by themselves, but omitting them forces the re-invocation of the scheduler at more and higher IIs, which does require significant additional time.

Figure 12 visualizes the impact of the different contributions on the benchmarks' total CGRA execution cycle count. We obtained these execution cycles using representative program inputs and a cycle-accurate simulator, with which we also validated the correctness of all compiled loops. The correlation between these results and those of Figure 11 confirms that the code quality metric of MII/II is a good proxy for real performance. Unlike what we observed for the MII/II numbers in Figure 11, however, we observe in Figure 12 that the omission of the fast mode leads to a small but noticeable difference in execution time. The reason is that in the total execution time, not only the II of the software-pipelined loops but also their number of pipeline stages and the length of their prologues and epilogues is relevant. Sometimes the fast scheduler produces better loops in that regard, and sometimes the accurate scheduler produces better ones.

5. CONCLUSIONS

In this article we presented a new list scheduler, called the BMS, for ADRES CGRA architectures. It builds on the existing state of the art but includes a set of modifications to make the scheduler produce better code at similar or even faster compilation times, in particular for more heterogeneous architectures. These modifications concern the backtracking heuristics, the priority function that determines the order in which operations are scheduled, and the cost functions used to guide the scheduler's router and to preserve resources for when they are truly needed.

Benchmark results on a set of loops and CGRA designs indicate that the BMS significantly outperforms the existing DRESC compiler in both code quality and compilation speed while supporting the same level of heterogeneity. On heterogeneous CGRA designs, for which the BMS is designed, the BMS generates significantly better code than DRESC. The BMS was also shown to at least compete with other techniques, such as HBEMS, RAMS, and GraphMinor, and there are indications that the BMS outperforms them when targeting more heterogeneous, more constrained CGRAs. Due to the limited number of benchmarks and a lack of shared benchmarks and shared CGRA designs, and due to differences in the used host CPUs, the significance of those indications remains to be seen.

REFERENCES

B. Bougard, B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins. 2008. A coarse-grained array accelerator for software-defined radio baseband processing. *IEEE Micro* 28, 4 (2008), 41–50.

F. Bouwens, M. Berekovic, B. De Sutter, and G. Gaydadjiev. 2008. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In *Proc. 3rd Conf. on High Performance Embedded Architectures and Compilers*. 66–81.

T. Cervero, A. Kanstein, S. López, B. De Sutter, R. Sarmiento, and J.-Y. Mignolet. 2008. Architectural exploration of the H.264/AVC decoder onto a coarse-grain reconfigurable architecture. In *Proc. of the Conf. on Design of Circuits and Integrated Systems*.

L. Chen and T. Mitra. 2014. Graph minor approach for application mapping on CGRAs. *ACM Trans. Reconf. Technol. Syst.* 7, 3 (2014), 21.

B. De Sutter, O. Allam, P. Raghavan, R. Vandebriel, H. Cappelle, T. Vander Aa, and B. Mei. 2010. An efficient memory organization for high-ILP inner modem baseband SDR processors. *Signal Process. Syst.* 61, 2 (2010), 157–179.

B. De Sutter, O. Coene, T. Vander Aa, and B. Mei. 2008. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *Proc. ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*. 151–160.

B. De Sutter, P. Raghavan, and A. Lambrechts. 2013. *Handbook of Signal Processing Systems* (2 ed.). Springer, Chapter Coarse-Grained Reconfigurable Array Architectures, 553–592.

B. De Sutter, D. Verkest, E. Brockmeyer, E. Delfosse, A. Vandecappelle, and J.-Y. Mignolet. 2009. Design and tool flow of multimedia MPSoC platforms. *Signal Process. Syst.* 57, 2 (2009), 229–247.

V. Derudder, B. Bougard, A. Couvreur, A. Dewilde, S. Dupont, L. Folens, L. Hollevoet, F. Naessens, D. Novo, P. Raghavan, T. Schuster, K. Stinkens, J.-W. Weijers, and L. Van Der Perre. 2009. A 200Mbps+ 2.14nJ/b digital baseband multi processor system-on-chip for SDRs. In *Proc. Symp. on VLSI Circuits*. 292–293.

C. Ebeling, L. McMurchie, S. Hauck, and S. M. Burns. 1995. Placement and routing tools for the Triptych FPGA. *IEEE Trans. VLSI Syst.* 3, 4 (1995), 473–482.

M. Hamzeh, A. Shrivastava, and S. B. K. Vrudhula. 2012. EPIMap: Using epimorphism to map applications on CGRAs. In *Proc. 49th Annual Design Automation Conf.* 1284–1291.

M. Hamzeh, A. Shrivastava, and S. B. K. Vrudhula. 2013. REGIMap: Register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *Proc. Annual Design Automation Conf.* 1–10.

C. Jang, J. Kim, J. Lee, H.-S. Kim, D. Yoo, S. Kim, H.-S. Kim, and S. Ryu. 2011. An instruction-scheduling-aware data partitioning technique for coarse-grained reconfigurable architectures. In *Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 151–160.

A. Kanstein, S. López, and B. De Sutter. 2007. Optimizing coarse-grain reconfigurable hardware utilization through multiprocessing: An H.264/AVC decoder example. In *Proceedings of the SPIE Conference: VLSI Circuits and Systems III*, Vol. 6590.

W. Kim, Y. Choi, and H. Park. 2013. Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures. *ACM Trans. Architec. Code Optim.* 10, 4 (2013), 1–24.

W. Kim, D. Yoo, H. Park, and M. Ahn. 2012. SCC based modulo scheduling for coarse-grained reconfigurable processors. In *Proc. Conf. on Field-Programmable Technology*. 321–328.

Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi. 2005. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7–11 March 2005, Munich, Germany*. 12–17.

M. S. Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 318–328.

A. Lambrechts. 2009. *Energy-Aware Datapath Optimizations at the Architecture-Compiler Interface*. Ph.D. Dissertation. Katholieke Universiteit Leuven.

A. Lambrechts, P. Raghavan, M. Jayapala, B. Mei, F. Catthoor, and D. Verkest. 2009. Interconnect exploration for energy versus performance tradeoffs for coarse grained reconfigurable architectures. *IEEE Trans. VLSI Syst.* 17, 1 (2009), 151–155.

G. Lee, K. Choi, and N. Dutt. 2011. Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *IEEE Trans. CAD Integr. Circ. Syst.* 30, 5 (2011), 637–650.

J. Llosa, E. Ayguadé, A. González, M. Valero, and J. Eckhardt. 2001. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. Comput.* 50, 3 (2001), 234–249.

S. A. Mahlke, D. C. Lin, W. Chen, R. E. Hank, and R. A. Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proc. Symp. Microarch*. 45–54.

B. Mei, B. De Sutter, T. Vander Aa, M. Wouters, A. Kanstein, and S. Dupont. 2008. Implementation of a coarse-grained reconfigurable media processor for AVC decoder. *Signal Process. Syst.* 51, 3 (2008), 225–243.

B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proc. Conf. on Field-Programmable Technology*. 166–173.

B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. 13th Conf. Field Programmable Logic and Application (FPL)*. 61–70.

D. Novo, T. Schuster, B. Bougard, A. Lambrechts, L. Van der Perre, and F. Catthoor. 2009. Energy-performance exploration of a CGA-based SDR processor. *Signal Process. Syst.* 56, 2–3 (2009), 273–284.

T. K. Oh, B. Egger, H. Park, and S. A. Mahlke. 2009. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, compilers, and tools for embedded systems (LCTES)*. 21–30.

J. Pager, R. Jeyapaul, and A. Shrivastava. 2015. A software scheme for multithreading on CGRAs. *ACM Trans. Embedded Comput. Syst.* 14, 1 (2015), 19.

H. Park, K. Fan, M. Kudlur, and S. A. Mahlke. 2006. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*. 136–146.

H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-S. Kim. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. Conf. on Parallel Architecture and Compilation Techniques*. 166–176.

H. Park, Y. Park, and S. A. Mahlke. 2009a. A dataflow-centric approach to design low power control paths in CGRAs. In *Proc. IEEE Symp. on Application Specific Processors*. 15–20.

H. Park, Y. Park, and S. A. Mahlke. 2009b. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proc. Symp. on Microarchitecture*. 370–380.

J. J. K. Park, Y. Park, and S. A. Mahlke. 2013. Efficient execution of augmented reality applications on mobile programmable accelerators. In *Proc. Conf. on Field-Programmable Technology*. 176–183.

Y. Park, H. Park, and S. A. Mahlke. 2009c. CGRA express: Accelerating execution using dynamic operation fusion. In *Proc. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*. 271–280.

R. B. Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. Symp. on Microarchitecture*. 63–74.

D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim. 2012. Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In *Proc. on Conf. Field-Programmable Technology*. 67–70.

T. Suzuki, H. Yamada, T. Yamagishi, D. Takeda, K. Horisaki, T. Vander Aa, T. Fujisawa, L. Van der Perre, and Y. Unekawa. 2011. High-throughput, low-power software-defined radio using reconfigurable processors. *IEEE Micro* 31, 6 (2011), 19–28.

T. Vander Aa, M. Palkovic, M. Hartmann, P. Raghavan, A. Dejonghe, and L. Van der Perre. 2011. A multi-threaded coarse-grained array processor for wireless baseband. In *Proc. 9th IEEE Symp. Application Specific Processors*. 102–107.

J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek. 2008. SPKM : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proc. 13th Asia South Pacific Design Automation Conf. (ASP-DAC)*. 776–782.

Received June 2015; revised December 2015; accepted February 2016