Programming Support for Reconfigurable Custom Vector Architectures*

Mehmet Ali Arslan
Lund University, Computer Science
mehmet_ali.arslan@cs.lth.se

Krzysztof Kuchcinski
Lund University, Computer Science
krzysztof.kuchcinski@cs.lth.se

Flavius Gruian
Lund University, Computer Science
flavius.gruian@cs.lth.se

Yangxurui Liu
Lund University, Electrical and Information Tech.
yangxurui.liu@eit.lth.se

ABSTRACT

High performance requirements increased the popularity of unconventional architectures. While providing better performance, such architectures are generally harder to program and generate code for. In this paper, we present our approach to ease programmability and code generation for such architectures. We present a domain specific language (DSL) for the programming part, and a constraint programming approach to scheduling with memory allocation. Our experiments on implementing a kernel extracted from a DSP application on an example reconfigurable custom architecture shows that it is possible to achieve performance close to hand-written machine code that is scheduled without memory allocation.

1. INTRODUCTION

Developments in computer architecture and implementation technology (FPGA, reconfigurable processors, etc.) leads to the development of custom architectures. These architectures are often designed to fulfill high performance requirements for a class of applications. However, this performance comes with a trade-off in programmability. Traditional compilers are not built to exploit the irregular structure and specific features in such architectures, and to adapt to the frequent changes in them. Therefore, using standard techniques and tools to compile from a high level language like C is not a compelling option.

A popular approach is to write machine code by hand. However, there are several problems with this approach. First of all, coding becomes extremely hard. The programmer has to select the instructions to implement the program. For architectures with VLIW and SIMD-like features, this means that the programmer needs to bundle small operations into instructions. To utilize the processor efficiently and increase throughput, the programmer also needs to come up with a schedule that parallelizes the code as much as possible, while respecting the resource and data storage limits. It can take many man-hours to write the machine code that corresponds to few lines in a high-level language. Secondly, the programmer needs to know the intricate details and complexities of the architecture, including but not limited to processor structure, memory layout, machine instructions, etc. Most of the time this level of information is limited to the architect only. And even for the architect, this overwhelming amount of information to be considered in programming and scheduling results in a tedious and error-prone process.

Our goal is to increase the programmability of such custom architectures without losing performance compared to the hand-written code (by the architect) by automating the program development process. As our target platform for this study, we selected a highly reconfigurable coarse grained architecture named EIT [1], which is built specifically for implementing MIMO algorithms efficiently. The architecture includes a pipelined vector processor, an accelerator for specific scalar operations and a specialized memory that enables access patterns matching the structure of the vector processor. To achieve our goal, we employ several techniques, as follows. We propose a domain specific language (DSL) that encapsulates the SIMD-like nature of the architecture and frees the programmer from instruction scheduling and memory allocation for data. The program written in this DSL is then compiled to an intermediate representation (IR) which is input to our scheduling procedure. Scheduling is combined with the memory allocation in a single constraint programming (CP) model, since these stages are intertwined with one another. Finally, the output is a schedule with memory allocation that contains all information needed by a code generator turning this schedule into machine code.

Generally in signal processing and specifically in MIMO applications, a large portion of the computational load comes from kernel programs that, are run many times for each piece of data [1]. This means, shortening the schedule for one kernel can drastically increase the overall performance. Therefore, aggressive optimization techniques targeting these kernels are beneficial even if they result in long compilation times. In this study we consider such kernels that can be represented as statically scheduled dataflow graphs without feedback edges.

The rest of this section introduces the architecture and the constraint programming technique briefly. It is followed by discussion on related work. In section 3, we describe the programming interface we implemented (DSL and its output, the IR) and the constraint model for scheduling and memory allocation. Section 4 presents the experiments and discusses the results followed by the conclusions and future work.

*This work has been supported by the Swedish Foundation for Strategic Research (SSF) as part of the High Performance Embedded Computing project (HiPEC).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA
Copyright 2015 ACM 978-1-4503-3404-4/15/02 ...$15.00.
http://dx.doi.org/10.1145/2712386.2712399

1.1 The EIT architecture

As previously mentioned, the architecture we target, as an example to reconfigurable custom architectures, includes a pipelined vector processor, an accelerator for specific scalar operations and a specialized memory that enables access patterns matching the structure of the vector processor. The implementation is based on a reconfigurable processor array framework presented in [2]. Figure 1 shows an overview of the micro-architecture of the processor. The processor consists of 6 processing (PE1−6) and 2 memory (ME1−2) elements interconnected via high-bandwidth low-latency links. According to the type of underlying operations, resource elements are partitioned in two. The vector block performs computationally intensive vector operations, while the accelerator part performs special operations such as division/square-root and CORDIC (COordinate Rotation DIgital Computer). Operation modes of these elements are specified in embedded configuration memories, which are re-loadable in every clock cycle. To ease runtime control of the whole processor, a master node (PE1) is responsible for tracking the overall processing flow. It also controls configuration memories based on instructions stored in ME1.

The vector block has three processing (PE2−4) and one memory (ME2) element, functioning as a multi-stage computation pipeline and a register bank, respectively. PE3 performs all vector operations. To concurrently compute multiple data streams, it is constructed from four homogeneous parallel processing lanes, each having four complex-valued multiply-accumulate (CMAC) units. This makes it possible to perform simultaneously four vector operations, that can have up to three operands, with vectors of four elements. To assist these vector computations, PE2 and PE4 pre- and post-process data to perform for example matrix Hermitian and result sorting. By combining these three processing elements, several consecutive data manipulations can be accomplished in one single instruction without storing and loading intermediate results. From the software perspective, the processing elements PE2−4 form a seven stage pipeline that does load (one stage), pre-processing (one stage), vector processing (two stages), post-processing (two stages) and write-back operations (one stage).

This execution scheme is similar to that of VLIW processors, but has additional flexibility for loading configurations into individual processing elements without affecting others, hence resulting in reduced control overhead.

The memory is organized in 16 *banks* to enable parallel access, needed for the vector processor. Banks are further grouped into *pages* to regulate the access to different *lines* in the banks. Each access is configured through descriptor registers assigned to each page. Two 4x4 matrices can be read and one 4x4 matrix can be written to the memory, simultaneously. Further details on the memory implementation can be found in [1]. We detail our abstraction of this implementation in section 3.4.

1.2 CP

In this paper we extensively use constraint satisfaction methods implemented in the constraint programming environment JaCoP [3]. In this section, we briefly introduce constraint programming and related constraints used in this work.

A *constraint satisfaction problem* is defined as a 3-tuple $\mathscr{S} = (\mathscr{V}, \mathscr{D}, \mathscr{C})$ where $\mathscr{V} = \{x_1, x_2, \ldots, x_n\}$ is a *set of variables*, $\mathscr{D} = \{\mathscr{D}_1, \mathscr{D}_2, \ldots, \mathscr{D}_n\}$ is a set of *finite domains* (FD), and $\mathscr{C}$ is a set of *constraints*. Finite domain variables (FDV) are defined by their domains, i.e. the values that are possible for them. A finite domain is usually expressed using integers, for example $x :: 1..7$. A constraint $c(x_1, x_2, \ldots, x_n) \in \mathscr{C}$ among variables of $\mathscr{V}$ is a subset of $D_1 \times D_2 \times \ldots \times D_n$ that restricts which combinations of values the variables can simultaneously take. Equations, inequalities and even programs can define a constraint. Each constraint is paired with a consistency technique to eliminate the infeasible values. These techniques can be complete (removing all infeasible values at once) or incomplete (removing a subset of infeasible values) depending on the choice of algorithms implementing them.

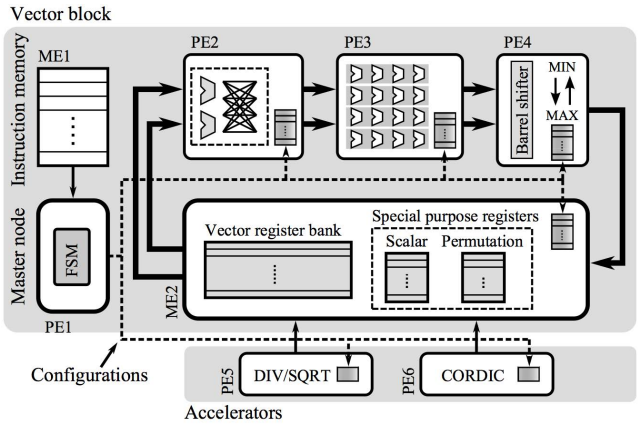

Figure 1: Micro-architecture of the vector processor, consisting of 6 PEs and 2 MEs. Solid and dashed lines depict data and control bus, respectively.

A global constraint on the other hand, combines several simpler constraints and handles them together. While semantically equivalent to the conjunction of these simpler constraints, a global constraint lets the solver exploit the structure of a problem by providing a broader view to it [4]. In this paper we use intensively two global constraints, namely `Cumulative`, `Diff2`.

`Cumulative` constraint [5] was originally introduced to specify the requirements on task scheduling on a number of resources. It expresses the fact that at any time the total use of these resources for the tasks does not exceed a given limit. It has four parameters: a list of tasks' starts, a list of tasks' durations, a list of amount of resources required by each task, and the upper limit of the amount of used resources. All parameters can be either domain variables or integers.

The `Diff2` [6] constraint is designed to model the placement of rectangles in two dimensional space in such a way that they do not overlap. It takes as an argument a list of rectangles and assures that for each pair of i, j $(i \neq j)$ of rectangles, there exist at least one dimension k where i is after j or j is after i. A rectangle is defined by a tuple $[O_1, O_2, L_1, L_2]$, where O_i and L_i are called the origin and the length of the rectangle in *i-th* dimension respectively. The `Diff2` constraint is used in this paper for defining constraints for resource binding, scheduling and lifetime binding for memory spaces (see section 3.4).

2. RELATED WORK

There are many aspects to code generation for custom architectures that relate to our work. Some of them are instruction selection, instruction scheduling and resource and register allocation. There is plenty of attention towards each of these topics, either in isolation or in combination, as in this work. Here we try to identify and report the most related ones.

Instruction selection and scheduling for a given processor or multiprocessor are complex problems known to be NP-complete. Special attention has been recently given to custom architectures that have complex instructions and non-regular instruction sets as well as possible reconfigurability of the processor under run-time. This makes it difficult to use well known compiler infrastructures, such

as LLVM [7]. An extensive survey about instruction selection by Blindell [8] is an invaluable text for further reading on the subject.

There are methods used for solving these problems optimally. Mixed integer programming (MIP), constraint programming (CP) or dynamic programming are common methods for mixed constrained versions of these problems.

Bednarski [9] explores optimal or highly optimized code generation techniques for in-order issue superscalar processors and various VLIW processors, using dynamic programming and integer linear programming (ILP). The dynamic programming method generates all possible solutions and searches for the optimal while shrinking the search space via pruning and compression techniques. Bednarski's work continues with investigating ILP formulation of the optimal code generation problem, again for VLIW architectures.

The work in [10] presents Unison, a code generator that addresses integrated global register allocation and instruction scheduling for architectures with VLIW capabilities, implemented with constraint programming. Input programs are represented in SSA (static single assignment) form. Merging instruction scheduling with register allocation in one model, Unison outperforms LLVM in most of the experiments presented and generates optimal code for a significant portion. Our approaches differ mainly in target architecture type (presence of vector processing capabilities) and the fact that our focus is on data memory allocation and access, while theirs is on register allocation.

Optimal basic block instruction scheduling for multiple-issue processors by Malik et al. [11] is another work using constraint programming. They schedule basic blocks from the SPEC 2000 integer and floating point benchmarks. The architectural model is VLIW-like, where several processing units run different types of basic instructions. Similar to our model, applications are represented as DAGs. Their target architecture does not have vector processing capabilities.

Another optimal method for instruction scheduling and register allocation is presented in [12], by Eriksson et al. They focus on clustered VLIW architectures and present an ILP method, combining instruction selection and scheduling with register allocation. For scheduling loops they employ modulo scheduling [13], which is a well established software pipelining technique that we use in this work as well, which is implemented in CP paradigm.

A heuristic approach for resource aware mapping on coarse grained reconfigurable arrays (CGRA) is presented in [14]. As in previously mentioned works, they also perform scheduling and register allocation in one single step. In scheduling, they employ modulo scheduling with backtracking. Reconfiguration costs are not mentioned in this work.

Our previous work [15] uses CP for instruction selection and scheduling for reconfigurable processor extensions that run very complex instructions and other architecture models such as VLIW processors as well as multicore RISCs. Instruction selection for complex instructions employs a custom global constraint for subgraph isomorphism. In this work we focus less on instruction selection and more on scheduling with memory allocation.

3. OUR APPROACH

The flow of the proposed programming support is depicted in figure 2. The application programmer is provided with a domain specific language written in Scala. When the application written in the DSL is run, an intermediate representation of the application is generated. This run can be used for debugging as well. The IR is the main input for the constraint programming model that generates a valid and efficient schedule and a corresponding memory allocation.

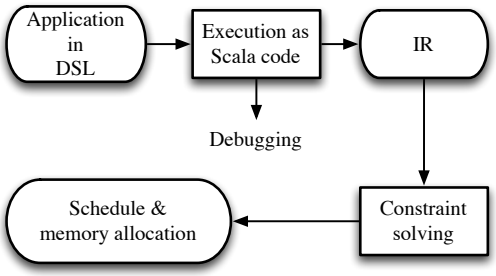

Figure 2: Programming support flow

This section explains describes the domain specific language (section 3.1) together with example code and intermediate representation. It continues with details of the constraint model for scheduling (section 3.3) and memory allocation (section 3.4).

3.1 Domain Specific Language

To ease programming, we devised a DSL that captures the SIMD-like nature of the architecture while leaving instruction scheduling and memory related details for the later stages of code generation. This way, the programmer is still able to write architecture-specific code without dealing with processor internal details, such as scheduling instructions in the pipeline without conflicts or where data is stored to and loaded from. The DSL provides architecture specific data types (matrix, vector, scalar) and handles necessary conversions between them both implicitly and explicitly.

The DSL is written as a library in Scala, and therefore the programmer is able to use any debugging tool available for Scala. This debugging is about the functional correctness of the code written in the DSL and not the machine code generated after scheduling. We have selected Scala since it is used quite commonly in implementation of embedded DSLs. It offers programming constructs such as pattern matching with case classes, traits, and combines functional programming with object-oriented programming (object-functional)[16].

Because of the reconfigurable nature of the architecture, the number of possible operations that can be run on the vector core is considerably large. To limit the operation set that is included in the DSL in our current implementation, we took a subset of the possible operations that are used in the MIMO applications and implemented them. Note that the modularity provided by Scala in the DSL implementation renders extending the operation set trivial, thus changing the operation set implemented by the DSL is not considered a problem.

Each operation in the DSL corresponds to an operation implemented in the architecture. This way, the programmer is still able to influence the details, which is desirable when programming such specialized architectures. Note that, this also means that the operations selected by the programmer during coding will be more or less the ones that are used in the machine code, even though these operations can be merged with others to build large instructions in later stages of code generation.

A simple matrix multiplication written in the DSL is given in listing 1. In this example we multiply a 4x4 matrix with its transpose. A matrix comprises four vectors of four scalars each. Instead of an explicit transpose operation, we access each jth vector in A as a column vector and get the dot product of it with the ith vector. This is done on line 16 with the operation v_dotP that takes two vectors and returns their dot product as a scalar.

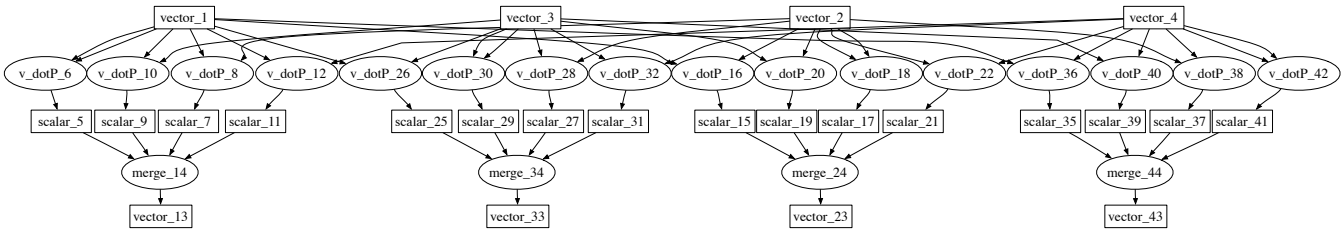

Figure 3: Intermediate representation of listing 1

Listing 1: Matrix multiplication in the DSL

```
1  //Hard coded input vectors
2  val v1 = EITVector(1,2,3,4)
3  val v2 = EITVector(2,3,4,5)
4  val v3 = EITVector(3,4,5,6)
5  val v4 = EITVector(4,5,6,7)
6
7  val A  = EITMatrix(v1,v2,v3,v4)
8
9  //Output buffer
10 val resultVectors = ListBuffer[EITVector]()
11
12 for(i<-0 until 4){
13   val scalars: Array[EITScalar] = new Array(4)
14   for(j<-0 until 4) {
15     //Vector dot product
16     scalars(j) = A(i) v_dotP A(j)
17   }
18   resultVectors.append(EITVector(scalars))
19 }
20 val res = EITMatrix(resultVectors.toList:_*)
```

The intermediate representation (IR) is generated from the code written in the DSL, depicted in figure 3.

3.2 Intermediate Representation

The IR is a dataflow graph represented as a directed acyclic graph (DAG) $G : (V, E)$ where V denotes the vertices (nodes) and E denotes the edges which represent the data dependency between the nodes. Nodes can be either *operation* nodes or *data* nodes. The graph is also bipartite. Every data node that is not an input of the application, is preceded by one operation node i.e. the operation that produces it. Similarly, every operation node is succeeded by a data node i.e. the data that is produced by it. For each node i, $cat(i)$ denotes the *category* it belongs to, which can be one of the following: *vector_op, matrix_op, scalar_op, index, merge, vector_data, scalar_data*. Additionally $op(i)$ annotates the operation for each operation node.

This dataflow graph is generated in XML format from the DSL code, which is later on input to the code generation tool chain. A visualization of the graph for the code in listing 1 is shown in figure 3. For clarity, the data nodes are drawn as rectangles while operations are ovals.

3.2.1 Data nodes

There are two types of data nodes shown in figure 3, namely *vector* and *scalar* nodes. These are actually all the data node types present in the IR. The *matrix* data type from the DSL is not included. Instead, data that is defined as a matrix in the DSL is expanded into four vector data nodes in the IR. The reason behind this decision is to keep the vectors as decoupled as possible to let the code generator decide on how to merge them, freely (see section 3.3). This can enable opportunities to improve the schedule. It also leads to an easier modeling for the memory related constraint regarding the vector data (see section 3.4).

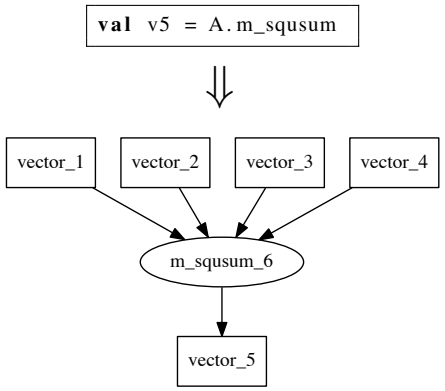

Figure 4: A matrix operation in the DSL an its IR

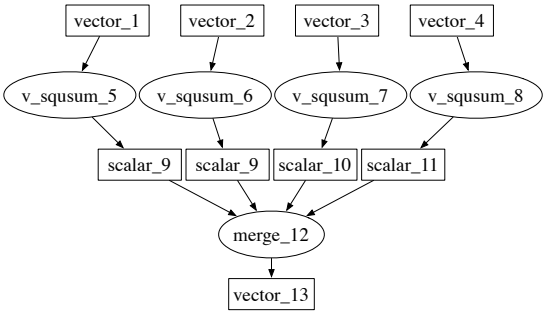

Figure 5: Vector implementation of `A.m_squsum` in figure 4

3.2.2 Operation nodes

The DSL implements a set of vector operations, e.g. *v_dotP* in listing 1, and each one of them corresponds to a single operation node in the IR, with the operation annotated as $op(i)$. These include the pre- and post-processing operations in the vector pipeline, such as masking and sorting. Operations defined on matrices result in a matrix operation node (see figure 4). In some of the cases it is possible to represent a matrix operation as four vector operations, each resulting in a scalar output. Figure 5 depicts such a vector implementation of the matrix operation in figure 4. However, the scalar outputs should then be merged to form the vector result, which is the proper output of the matrix operation. Using the matrix versions of such operations removes these *merge* nodes and decreases the total number of nodes generated.

Apart from the nodes mentioned above, there are also nodes for scalar operations, such as the square root operation that runs in an accelerator separate from the vector core. Merging (as can be seen in figure 3) and indexing vectors are also included as nodes in the IR.

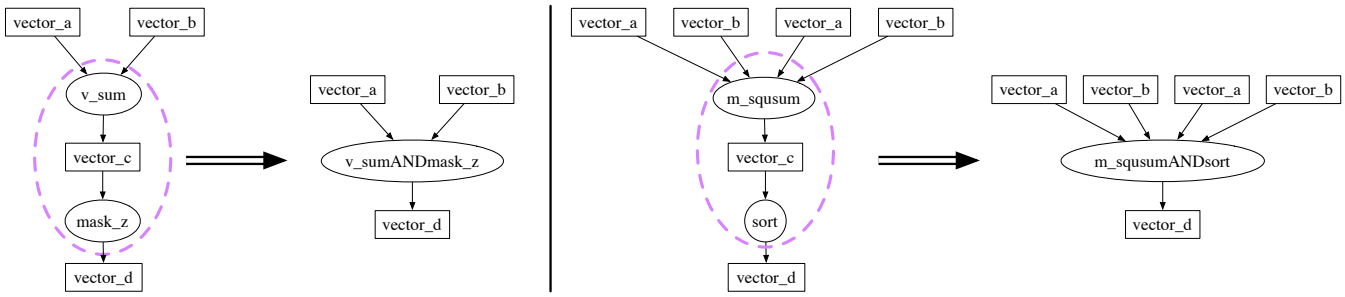

Figure 6: Merging examples. To the left: Merging a vector operation with a pre-processing operation. To the right: Merging a matrix operation with a post-processing operation when the post-processing is done on the vector output.

3.3 Scheduling an application

In this phase, scheduling is responsible for the following:

- assigning a start time for each node
- finding a configuration for the vector pipeline on each cycle
- minimizing the schedule length

While realizing these goals, there is a set of constraints that has to be respected. These can be categorized as following:

- precedence constraints
- resource constraints

In the rest of this section we explain each goal with its respective constraints.

We assign three finite domain variables (FDV) for each node: s, l, d. s_i will denote the start time variable for node i while l_i denotes its *latency* and d_i its *duration*. Latency represents the time that passes from the start time of the operation until its output is ready to use. Each operation occupies the resource it is run on for some time, which is denoted by duration. For data nodes, both of these variables are set to zero.

3.3.1 Precedence constraints

Precedence between two nodes is represented in the IR as an edge between them. An edge from node i to node j means that i has to be finished before j can start. Since the dependency here is a data dependency, it is the *latency* that has to be taken into account and not *duration*. This relation is embodied in constraint (1).

$$\forall (i,j) \in E : s_i + l_i \leq s_j \tag{1}$$

The vector pipeline consists of seven stages, including pre-, core- and post-processing, that amounts up to a latency of 7 clock cycles.

In order to decrease complexity, we model the pipeline as a whole, instead of modeling its stages one by one. This results in a discrepancy between the IR and the constraint model, since operations that would be run in the same pipeline (which follow the pre-, core-, and post-processing pattern) are represented with one node each. We remove this discrepancy by merging vector operations that follow the pre-, core-, and post-processing pattern into one node, whenever possible. This is carried out on the IR before the scheduling starts. Two such examples are depicted in figure 6. This decreases the complexity in two ways. First, the number of nodes is decreased, which almost always results in an easier problem. Second, and more importantly, after this merging, we do not need to model each pipeline stage separately. We can now assume that each vector operation has a latency of 7 clock cycles, i.e. the latency of the pipeline.

3.3.2 Resource constraints

The vector processor is capable of running up to four vector operations or one matrix operation, simultaneously. If we see the processor as having four vector lanes of computation, we have to make sure that we do not overload them at any time point in the schedule. For this purpose CP offers a well-studied global constraint named `Cumulative`, that is used commonly for task scheduling problems [5] (see section 1.2).

For the group of operations nodes that are run in the vector core we impose constraint (2). The parameters are their start times, durations, number of resources (lanes) they occupy (which is represented with r_i for node i), respectively. The last parameter (*nLanes*) is the number of available resources, which is in this case four. This grouping includes all vector and matrix operations. The difference between a vector and a matrix operation is that a vector operation occupies only one lane ($r = 1$), while a matrix operation occupies all four lanes ($r = 4$) i.e. nothing else can run in the vector processor at the same time. The duration of either operation is 1 clock cycle ($d = 1$).

$$\begin{aligned} &\texttt{Cumulative}(S,\ D,\ R,\ nLanes) \\ &where: \\ &S = [s_i \mid cat(i) = vector\_op \vee cat(i) = matrix\_op] \\ &D = [d_i \mid cat(i) = vector\_op \vee cat(i) = matrix\_op] \\ &R = [r_i \mid cat(i) = vector\_op \vee cat(i) = matrix\_op] \end{aligned} \tag{2}$$

Different lanes of the vector processor can not be configured to execute different operations simultaneously. Therefore, we need to ensure that at any given time, the operations in different lanes are the same. This is done by differentiating the start times of each vector operation pair (see constraint (3)).

$$\forall (i,j) \mid cat(i) = cat(j) = vector\_op \wedge op(i) \neq op(j) : \\ s_i \neq s_j \tag{3}$$

In a similar way, we impose one `Cumulative` constraint for the scalar processor and one for the part of the architecture that is responsible of indexing and merging (that we see as just another resource). Since they can run only one operation at a time, the resource limit for these constraints is 1.

3.3.3 Scheduling data nodes

So far we only discussed the scheduling of the operation nodes, however data nodes should also be scheduled. The relation between these nodes and the memory is explained in the next section. Here we only detail the part pertaining to the schedule.

We assume that the inputs to the application are ready from the start. Hence, any data node without any predecessors get the start

time zero. Any other data node starts, when the operation that produces it finishes and its latency is passed. Constraint (4) captures this relation where $pred(i)$ denotes the predecessor of node i in the graph. Note that each data node (except the application inputs) has only one predecessor, namely the operation that produces it.

$$\forall i \mid cat(i) = vector\_data \vee cat(i) = scalar\_data : \\ s_i = s_{pred(i)} + l_{pred(i)} \tag{4}$$

3.3.4 Minimizing the schedule length

Minimizing the schedule length is achieved by using the latest completion time ($s_i + l_i$) among all nodes as the objective function of the optimization process (see section 3.5).

$$obj = max_{i \in V} \ (s_i + l_i) \tag{5}$$

3.4 Memory

As the target architecture employs a special memory structure for the vector data, we dedicate this section to briefly present our memory layout abstraction (depicted in figure 7), the rules that regulate the access to it, and our constraints that implement these rules. Note that, because of its special structure, our focus is the vector memory and for scalar data, we assume optimal allocation and access.

The memory consists of 16 *banks* to enable parallel access. Four banks construct a memory *page*. Each page employs an access configuration, to program the access to the banks in that page. The smallest addressable memory unit in this abstraction is the *slot*, which holds a vector. If we were to address slots with the *(bankNo, slotNo)* pair, all the slots with the same *slotNo* build a *line*.

Each bank can be accessed once for reading and once for writing in each clock cycle, with a maximum of eight vectors (two matrices) read and four vectors (one matrix) written to the entire memory. This means that several slots can be accessed simultaneously only if they are in different banks. Also, since memory access reconfiguration is very costly in the target architecture, to limit and regulate access, simultaneous access to slots in a page is only allowed when the slots reside in the same line.

To explain with an example, we consider three matrices, whose vectors are allocated in different ways, in a small memory with three slots per bank as in figure 8. Matrix A can not be accessed in one cycle, since vectors A_1 and A_3 reside in the same bank, as well as A_2 and A_4. Matrix B can not be accessed in one cycle either, because of its vectors B_3 and B_4 that reside in the same page but not in the same line. To access them, the access to page 3 (banks 8-11) has to be reconfigured at least once. Matrix C, on the other hand complies with all the constraints, and can be accessed in one cycle.

To implement the access restriction constraints, we introduce the FDVs $slot_i$, $line_i$ and $page_i$ for each vector data node i. These variables actually represent different views of the same information: the placement of vector i in memory. $slot_i$ would be enough to represent this. $line_i$ and $page_i$ are defined mainly for modeling convenience. Slots are enumerated in a linear fashion i.e. the first slot in the first bank is labeled 0, the first slot in the second bank is labeled 1, etc. Correspondingly, the second slot in the first bank is labeled 16 while the second slot in the second bank is labeled 17, and so on.

The connections between $slot_i$, $line_i$ and $page_i$ for node i are given in the constraint group (6) where $nOfBanks = 16$ and $pageSize = 4$ in our particular architecture.

$$\forall i \mid cat(i) = vector\_data : \\ line_i = slot_i / nOfBanks \\ page_i = (slot_i \bmod nOfBanks) / pageSize \tag{6}$$

By their inputs and outputs, vector operations define the accesses to the vector memory. Therefore, to constrain the simultaneous access patterns, we need a two-fold control. First, we have to constrain the allocation of the inputs of each vector core operation, since the inputs are accessed simultaneously. Second, any vector core operations that are scheduled to run simultaneously will access the memory simultaneously as well, both for their inputs and their outputs. So, the allocation of those inputs and outputs should be constrained as well. The first part is captured in constraint (7).

$$\forall i \mid cat(i) = vector\_op \ \vee \ cat(i) = matrix\_op : \\ \forall (d,e) \mid d,e \in pred(i) \ \wedge \\ cat(d) = cat(e) = vector\_data : \\ page_d = page_e \implies line_d = line_e \tag{7}$$

The second part is achieved with checking accesses of vector operation pairs that are of the same type and are scheduled at the same time (see constraints (8) and (9)).

$$\forall i,j \mid cat(i) = cat(j) = vector\_op \ \wedge \ s_i = s_j : \\ \forall (d,e) \mid d \in pred(i) \ \wedge \ e \in pred(j) \ \wedge \\ cat(d) = cat(e) = vector\_data : \\ page_d = page_e \implies line_d = line_e \tag{8}$$

$$\forall i,j \mid cat(i) = cat(j) = vector\_op \ \wedge \ s_i = s_j : \\ \forall (d,e) \mid d \in succ(i) \ \wedge \ e \in succ(j) \ \wedge \\ cat(d) = cat(e) = vector\_data : \\ page_d = page_e \implies line_d = line_e \tag{9}$$

To use the memory space economically, we need to reuse the slots when their data is no longer needed. To decide on when a slot can be used, we define *lifetime*s for each data node. The lifetime of a data node ($life_i$) is defined as the interval between the start time of the node itself and the start time of the latest operation that uses it. This relation is captured in constraint (10) where $succ(i)$ denotes the successors of node i.

$$\forall i \mid cat(i) = vector\_data : \\ life_i = \max U_i - s_i \\ where : \\ U_i = [s_o \mid o \in succ(i)] \tag{10}$$

In a correct memory allocation, lifetimes associated with slots do not overlap. Using the fact that slots are enumerated linearly, we can model the memory allocation with reuse, as the non-overlapping rectangles problem. s_i and $slot_i$ become the horizontal and vertical origins, respectively, while $life_i$ denotes the length of the rectangle i. Height is set as 1 since each vector occupies one slot only. This way, we can use the highly efficient `Diff2` global constraint which is described in detail in section 1.2 as seen in constraint (11).

$$\texttt{Diff2}(S, SL, L, ones) \\ where : \\ S = [s_i \mid cat(i) = vector\_data], \\ SL = [slot_i \mid cat(i) = vector\_data], \\ L = [life_i \mid cat(i) = vector\_data] \tag{11}$$

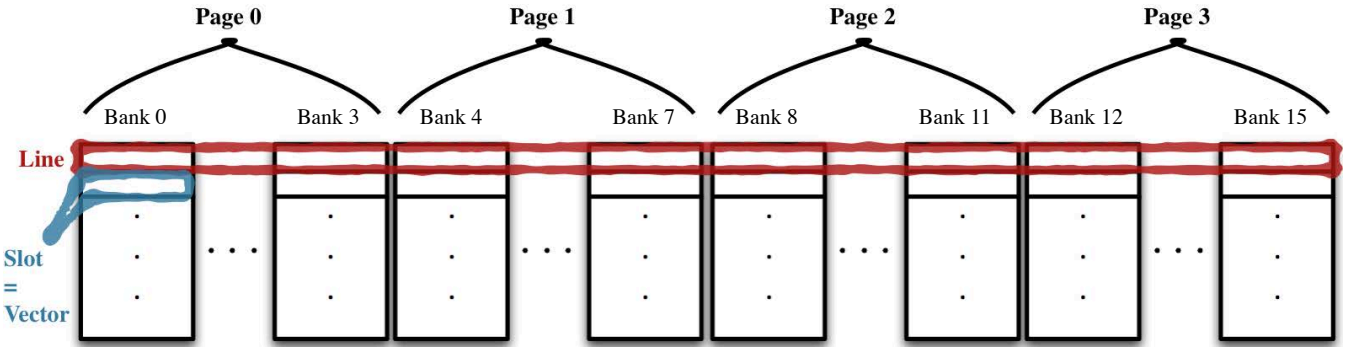

Figure 7: Memory layout abstraction. Memory is organized in pages, lines and slots.

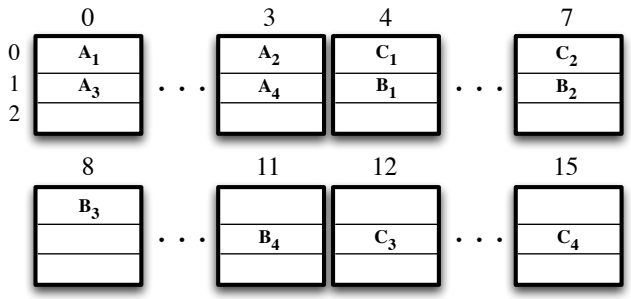

Figure 8: Memory access examples. Only C can be accessed in 1 cycle

3.5 Search space heuristics

The optimization goal is finding the shortest schedule (or one such schedule in case several shortest schedules exist). The completion of node i is defined as $s_i + l_i$ and minimizing the maximum of this completion for all nodes gives the shortest schedule.

As most consistency techniques are not complete (see section 1.2), the constraint solver needs to search for possible solutions, commonly by picking a variable that has not been assigned to a value yet, and setting it to a value in its domain. As long as the constraints are correct, any variable selection method (called *heuristic* in CP terminology) leads to a valid solution, eventually. However, depending on the problem size, the search space may grow exponentially and lead to very long search times. In order to decrease the search time, we need to devise a search strategy that defines the variable and value selection heuristics. In a previous work [15], we devised such a search strategy for a very similar problem. We briefly describe it in the following.

Even though the constraint model is unified, we divide the search into three sequential phases and every phase has a set of variables to pick from:

1. Scheduling the operation nodes, searches on
   S_{ops}: operation node start times
2. Scheduling the data nodes, searches on
   S_{data}: data node start times
3. Memory allocation, searches on
   SL: slots

The general idea behind this division is to start with the most influential decisions and end with the most trivial ones. This way, each time the solver needs to make a decision (i.e. pick a variable and a valid value for it) for triggering constraints to prune values in variable domains, it will pick the decisions that propagate more information. The first two phases are tasked with the optimization, namely to minimize the schedule length. The last phase takes the schedule result from the previous phases and searches for a valid assignment to the slot variables only. At the end of third stage we have a schedule with a valid memory allocation. All three phases are bundled together as a branch-and-bound search with backtracking, for finding the schedule with the minimum length.

Application properties	schedule length (cc)	#slots available	#slots used	opt. time (ms)
$\|V\| = 143$, $\|E\| = 194$	173	64	33	1854
$\|Cr.P\| = 169$,	173	32	28	1844
# v_data = 49	173	16	16	1813
	173	10	10	1835

Table 1: Scheduling QR decomposition on the EIT architecture

4. EXPERIMENTS

To evaluate our method, we implemented and scheduled a kernel, which is a part of a larger DSP application, and experimented with different ways of overlapping iterations of this kernel, to increase utilization and throughput. In the following, we first introduce the target application, QR decomposition (QRD), that is used in most of our experiments. After this we report our experiments on scheduling one instance of QRD, and discuss the results which displayed poor processor utilization. The rest of the section briefly introduces several methods to overlap several iterations of the same application to alleviate poor utilization, and presents our experiments using these methods on QRD and a pair of other applications.

4.1 Target application

As the main target application, we focused on the Modified Gram-Schmidt (MGS) based minimum mean squared error (MMSE) QRD algorithm, which is used as part of the pre-processing in data detection in multiple-input multiple-output (MIMO) systems [17]. The implementation in DSL was carried out by one of the designers of the target architecture, based on the MMSE-QRD algorithm given in [1].

4.2 Scheduling one iteration

With the model explained so far, we have scheduled a QRD with memory allocation.

In table 1, the results of scheduling QRD with different memory sizes (available number of slots) is shown. The leftmost column includes general properties of the IR graph and the resulting constraint model. As seen in the third column, the memory size,

# iterations = 12	Manual	Automated
Schedule length (cc)	460	540
# reconfigurations	18	24
# reconfigs/# iter.	1.5	2
Throughput (iter./cc)	0.026	0.022

Table 2: Overlapping iterations with focus on limiting the number of reconfigurations

i.e. the number of available slots, is parameterizable in our model. The reason why the schedule length stays the same with changing memory size is explained by examining the length of the critical path ($|Cr.P|$ in the table). As the $|Cr.P|$ is almost identical to the schedule length, it dominates the optimization process. This also means that memory size is a secondary issue for this problem. For fewer than 10 available slots, the solver timed out without finding a solution when the size was 9, and failed when it was 8, denoting that no solution exists for 8 slots.

The schedule length (which is the same for all experiments in table 1) is minimal, based on the given memory size and the algorithm implementation in the DSL. There are many different ways to express the same algorithm in the DSL, and these different expressions may result in different graphs, which in turn may result in different schedules.

Although getting an optimal schedule is valuable, this schedule includes a lot of "gaps", mainly because of the data dependencies between vector operations. Since each vector operation has a latency of 7 clock cycles, a vector operation that takes the output of another vector operation as its input has to wait for those 7 cycles. If there are no other vector operations in the application that can be run in this interval, the vector processor stays idle. If this repeats often, the processor becomes heavily under-utilized. Some techniques to overcome this are presented in the next section.

4.3 Scheduling more iterations simultaneously

To increase utilization, it is a common practice to schedule several iterations/copies of the same application. The idea is to schedule an available operation from another iteration when the processor is idle because of a data dependency explained above, and increase utilization and overall throughput (at the possible expense of latency).

There are several possible ways to implement this kind of simultaneous execution of iterations, with varying results in throughput, latency and reconfiguration complexity [18].

A simple ad-hoc technique, often employed by the architecture designers when they manually program the vector processor is the following two-phase process we refer to as *overlapped execution*. First the instructions for a single iteration are selected and ordered, usually with the objective of minimizing the number of effective (non-*nop*) instructions. Then the overlapped schedule is obtained by executing in sequence the same corresponding instruction from a given number M of iterations. Once all kth instructions, from all M iterations, have been scheduled, the execution advances to executing all $(k+1)$th instructions, and so forth. Note that this effectively masks the pipeline latency, when the number M of iterations is larger than the number of stages.

Besides being a computationally simple solution, this approach is also an efficient way of decreasing the number of reconfigurations needed. A reconfiguration is needed when two different types of instructions follow each other, which here only happens between every kth instruction of the last iteration and every $(k+1)$th iteration of the first iteration. This means that the number reconfigurations needed is limited to the number of instructions.

We used this technique based on our initial schedule, and compared the results to the manual implementation and scheduling, as shown in table 2. The margin between the automated and the manual scheduling is close to 20%. It is likely and reasonable that the manual implementation is more efficient than the translation from the DSL, especially considering the instruction selection. However, note that the manual implementation does not include memory allocation and involves tedious man-hours to complete. Especially, creating a conflict-free pipeline schedule by hand is a capriciously difficult and error-prone task.

The most important negative side effect of this pipelining approach is that it postpones all output to the last bit of the schedule, where every last operation of every iteration is scheduled one after the other. While the average throughput is not affected, this might lower the quality in streaming applications because of its bursty throughput instead of a stable one. Also, since all output is postponed to the end of the schedule, the sizes of the buffers needed to store the intermediate results will be large.

Another way of executing several iterations simultaneously makes use of modulo scheduling [13], which is a technique used very often in scheduling loops in VLIW-like architectures [19]. Modulo scheduling revolves around finding a schedule that initiates iterations as soon as possible, taking into account dependencies and resource constraints, and also repeating regularly with a fixed interval (also called *initiation interval (II)*). The net result of this technique is a more efficient use of the resources, thus yielding a better throughput, calculated as $1/II$.

We pipelined our initial schedule using modulo scheduling, modelled as another constraint satisfaction problem (CSP). First, we employed a model that finds a schedule with minimum possible II without taking into account the reconfiguration overhead. The reconfigurations are added and recorded only in a post processing steps. To contrast, we also implemented a model that does include the reconfigurations in the optimization process. Table 3 gathers the results of these two techniques for QRD and two other, less complex applications. For the sake of brevity, the details of the constraint model are omitted.

In case of QRD, where there are many reconfigurations to consider, the model excluding the reconfigurations proved to be an easier problem to solve. However the one that includes them provides a better throughput since reconfigurations have to be added to the minimum II found in the first model to get the actual II. The tradeoff is the optimization time required to find the modulo schedule for the second method. The solver times out after searching for the optimal solution for 10 minutes. For harder problems the execution time of the solver can grow and degrade the solution quality. In table 3, the cell for execution time for the QRD denotes the time elapsed to find the solution with $II = 46$, before timeout.

Compared to ad hoc method mentioned previously (*overlapped execution*), the model including the reconfigurations finds a schedule that performs just as well (see the throughput for the automated scheduling table 2), in terms of average throughput. Furthermore, modulo scheduling provides a stable throughput while overlapping iterations suffer from burstiness.

The two additional applications in our experiments are auto regression filter (ARF), and matrix multiplication (MATMUL, implemented as in listing 1). ARF was modified to work on vectors as basic units instead of scalars, in order to exploit the vector capabilities of the architecture. The model including the reconfigurations displays a similar improvement in throughput as QRD. However, this model results in a penalty in execution time. MATMUL uses only one type of operation throughout the application, therefore no reconfiguration is needed after the first instruction.

Application	($\|V\|, \|E\|, \|Cr.P\|$)	optimization excluding reconfigurations				optimization including reconfigurations		
		initial II (cc)	# rec.	actual II (cc)	throughput (iter./cc)	II (cc)	throughput (iter./cc)	optimization time (ms)
QRD	(143, 194, 169)	32	23	55	0.018	46	0.022	3055
ARF	(88, 128, 56)	16	16	32	0.031	24	0.042	80061
MATMUL	(44, 68, 8)	4	1	4	0.250	4	0.250	2135

Table 3: Pipelining with focus on limiting the number of reconfigurations

Note that with the assumption that there is enough memory for storing the data for all the iterations that are overlapped, memory allocation boils down to repeating the allocation of the original schedule for each iteration, with a certain offset.

5. CONCLUSIONS AND FUTURE WORK

In this work, we provided programming support for a custom reconfigurable architecture, that combines features similar to VLIW and SIMD with a specialized memory layout. The programming support consists of a DSL, an instruction scheduler and memory allocator that makes efficient use of the custom nature and features of the architecture. Our results show that our method can be useful for programming similar architectures, providing ease of programming and performance close to hand-written machine code.

We plan to continue this work by targeting other vector architectures including commercial processors, and more complex applications. Including reconfigurations in the constraint model for modulo scheduling proved to be a challenge that we also would like to investigate further.

6. REFERENCES

[1] C. Zhang, "Dynamically Reconfigurable Architectures for Real-time Baseband Processing," Ph.D. dissertation, Lund University, 2014. [Online]. Available: http://lup.lub.lu.se/record/4406448/file/4406451.pdf

[2] C. Zhang, L. Liu, and V. Öwall, "Mapping Channel Estimation and MIMO Detection in LTE-Advanced on a Reconfigurable Cell Array," in *IEEE International Symposium on Circuits and Systems (ISCAS), 2012, 2012-05-20/2012-05-23*. IEEE, 2012.

[3] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 355–383, Jul. 2003.

[4] W.-J. van Hoeve and I. Katriel, *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence. Elsevier Science, 2006, ch. Global Constraints.

[5] A. Aggoun and N. Beldiceanu, "Extending chip in order to solve complex scheduling and placement problems," *Mathematical and Computer Modelling*, vol. 17, no. 7, pp. 57 – 73, 1993.

[6] N. Beldiceanu and E. Contejean, "Introducing global constraints in CHIP," *Journal of Mathematical and Computer Modelling*, vol. 20, no. 12, pp. 97–123, 1994.

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[8] G. S. Hjort Blindell, "Survey on instruction selection: An extensive and modern literature study," KTH Royal Institute of Technology, Stockholm, Sweden, Tech. Rep., October 4 2013, ISBN: 978-91-7501-898-0.

[9] A. Bednarski and C. Kessler, "Integer Linear Programming versus Dynamic Programming for Optimal Integrated VLIW Code Generation," in *12th Int. Workshop on Compilers for Parallel Computers*, 2006.

[10] R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte, "Combinatorial spill code optimization and ultimate coalescing," in *ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES' 14, June 12–13 2014.

[11] P. v. B. Abid M. Malik, Jim McInnes, "Optimal basic block instruction scheduling for multiple-issue processors using constraint programming," in *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, 2005.

[12] M. Eriksson and C. Kessler, "Integrated modulo scheduling for clustered vliw architectures," in *High Performance Embedded Architectures and Compilers*, ser. Lecture Notes in Computer Science, A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, and T. Ungerer, Eds. Springer Berlin Heidelberg, 2009, vol. 5409, pp. 65–79.

[13] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 318–328.

[14] G. Dimitroulakos, S. Georgiopoulos, M. D. Galanis, and C. E. Goutis, "Resource aware mapping on coarse grained reconfigurable arrays," *Microprocess. Microsyst.*, vol. 33, no. 2, pp. 91–105, Mar. 2009.

[15] M. A. Arslan and K. Kuchcinski, "Instruction selection and scheduling for DSP kernels," *Microprocessors and Microsystems*, vol. 38, no. 8, Part A, pp. 803 – 813, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933114000520

[16] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*, 1st ed. USA: Artima Incorporation, 2008.

[17] P. Luethi, A. Burg, S. Haene, D. Perels, N. Felber, and W. Fichtner, "VLSI Implementation of a High-Speed Iterative Sorted MMSE QR Decomposition," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, May 2007, pp. 1421–1424.

[18] M. A. Arslan, F. Gruian, and K. Kuchcinski, "A comparative study of scheduling techniques for multimedia applications on SIMD pipelines," submitted for publication.

[19] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein, "Software pipelining showdown: Optimal vs. heuristic methods in a production compiler," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96. New York, NY, USA: ACM, 1996, pp. 1–11.