Code Generation for Custom Architectures using Constraint Programming

Arslan, Mehmet Ali

2016

Document Version:
Publisher's PDF, also known as Version of record

Code Generation for Custom Architectures using Constraint Programming

Mehmet Ali Arslan

Doctoral Dissertation, 2016

Department of Computer Science
Lund University

978-91-7753-024-4 (Press)
978-91-7753-025-1 (PDF)
ISSN: 1404-1219
Dissertation 54, 2016
LU-CS-DISS: 2016-06

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden
Email: mehmet_ali.arslan@cs.lth.se

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2016

In loving memory of Mehmet Hadi Arslan (1982 - 2013),
and for Yusuf to outmatch...

CONTENTS

POPULAR SCIENCE SUMMARY

The computation power we expect from the various smart devices we use keeps increasing. Not only do we want faster devices but also less power hungry and energy efficient devices, both for the environment and our personal convenience (remember that "mobile phone" attached to a power plug at all times?).

One way of addressing this demand is to build custom processor architectures that focus on a specific application domain and meet specific demands such as limited power budget, bandwidth requirements, and chip area. As a wise woman once said, "there is no such thing as a free lunch" and in contrast to general purpose processor architectures, these architectures tend to end up notoriously hard to program. This is because of the customization of the hardware to a level that it becomes hard and inefficient to use tools and languages available for general purpose processors. So much so, that they quite often become solely programmable in the machine language specific to the architecture. This means many expert-hours spent in manual translation of relatively simple programs into machine code, rendering the architecture hardly usable by anyone else than the architect.

This thesis is the result of our effort to increase the programmability of such custom architectures through automatic code generation, without losing performance compared to code written manually by the architect. Automatic code generation for general purpose architectures is a well studied research area and there exist many straightforward techniques. However, modeling code generation for custom architectures is complicated by the restrictions and constraints of the architectures, and performance requirements that need to be met for the targeted applications.

Constraint programming is a programming paradigm that fits problems defined naturally by constraints and relations between entities. Here, a problem is formulated as a series of constraints over placeholder variables (much like the empty

squares in sudoku) and solved by a constraint solving engine. The solving engine eliminates the infeasible values for each placeholder variable step-by-step, until a solution with each variable assigned to a value is found. As the capabilities and restrictions on the architectures, and the requirements on the applications we target can easily be translated into constraints, we choose constraint programming as our tool for modeling code generation for custom architectures.

Throughout the thesis we demonstrate the effectiveness of our method by comparing to theoretical or practical bounds and code written manually by the architect. The frameworks we present make the architectures easier to program by letting the programmer write in a higher level language than the specific architecture's machine language. Our experiments show that the machine code generated by our frameworks are competitive with the state of the art.

ACKNOWLEDGMENTS

Have We not opened up thy heart,
and lifted from thee the burden
that had weighed so heavily on thy back?
And [have We not] raised thee high in dignity?
And, behold, with every hardship comes ease:
verily, with every hardship comes ease!
Hence, when thou art freed [from distress], remain steadfast,
and unto thy Sustainer turn with love.

Al-Inshirah, Qur'an
(As rendered by Muhammad Asad)

Five years is a long time. Several times I doubted I would make it to the "Acknowledgments". But here we are. This would not be possible without the help I got from quite many people.

I am forever grateful to Prof. Krzysztof Kuchcinski for his utmost patience and almost fatherly support during these years, besides the excellent supervision he provided me. I was very lucky to have Dr. Flavius Gruian as a friend and supervisor, with his extreme-precision-feedback and bulletproof tolerance to all the "things" I came up with during these five years. Also, between you and me, he is funny.

A special thanks to Dr. Jörn W. Janneck for his supervision and timely brutal honesty, which helped me get back on track when I felt lost the most. Thanks to Prof. Pierre Flener for his contagious enthusiasm about constraint programming. Without his introduction, I would not find my favorite topic in computer science.

Collaboration in writing a paper can be very tricky. Big thanks to Andréas Karlsson, Chenxin Zhang, Yangxurui Liu and Essayas Gebrewahid for making it so easy.

Another big thanks to the administrative staff in the department, who put up with my silly questions, annoying issues about my visa, and many other things... Thanks to all Pakistani and non-Pakistani colleagues in the department for creating the inclusive and welcoming atmosphere.

Thanks to Çağdaş Aydın for introducing me to computer science, and saving me from choosing economy or something as boring. Thanks to Prof. Muhammet Toprak and his wonderful family for helping me in my first contact with Sweden, cushioning the culture shock. Thanks to Esat Arslan for being my mentor in life. Eternal thanks to my brother Ihsan Arslan and his family for being there whenever I needed them. And to Mehdi, for his love, wisdom and care.

My friends educated me throughout my life, I would not be this self without them. The list is long, but they know who they are, so thank you.

I have a really huge family back home in Turkey, and I know all of them have been rooting and praying for me since I decided to move to Sweden for studying. In contrast, I have a relatively short space here, so while I am grateful to all of them, I will single out my singular mother here, who raised me alone, with lots and lots of unconditional love. I owe her everything and if this thesis makes her proud, then I am proud of myself.

My last year in PhD was by far the best one, as this is the one I met my love and best friend, Lina Dahlman. I am so very lucky to have you. On that note, thank you Han Solo; yes you died, but it was definitely worth it.

As I hinted with the quote above, all this gratitude originates from and returns to The Most Gracious, The Dispencer of Grace - Allah. I am humbled by the countless blessings poured over me...

1 INTRODUCTION

Embedded systems are a special class of computing systems with very specific requirements on performance, cost and power/energy consumption. Applications that are targeted by embedded systems are getting more and more computation hungry. Many of these applications, especially those in telecom and signal processing, require high throughput (processed data per time unit) preferably with low latency, so that when the result of a requested computation is produced, it is still relevant to the user. On the other hand, the power and area budget is limited, for reasons varying from battery lifetime and pocket space to environmental concerns. These requirements are often addressed via special design and architectural choices. In particular, *custom processor architectures* (a.k.a. customizable processors) are often employed. To meet the requirements, custom architectures are tailored to fit the target application domain. This includes providing the amount of *instruction level parallelism* (the number of instructions the processor can run simultaneously) that is sufficient for the applications, implementing certain critical operations in hardware, moving non-critical functionality from hardware to software (e.g. by emulation) to increase clock speed and customizing the memory structure to fit the common data access patterns [1, 2, 3].

Many applications that are designed for embedded systems (e.g. image processing, telecom, surveillance) are inherently *parallel*. This means that, even though they are defined as a series of sequential instructions, a significant amount of those instructions are independent of each other and thus can be run simultaneously. A specific type of parallelism that commonly occurs in digital signal processing (DSP) and multimedia applications is *data parallelism*, where the same operation is executed on many data. This type of parallelism is usually supported by Single Instruction Multiple Data (SIMD) instructions, which enable processing vectors of data instead of single elements. However, having a vector processor without the data management that enables vector access would only make the processor wait for the data. Therefore SIMD instructions require high-bandwidth memory architectures to feed the processor with enough data.

From an embedded system architect's point of view, architectural customization may be the only step necessary to meet the specific requirements from the application and user domain. However, the target application has to be programmed in a custom manner as well, to reap the benefits of having special hardware. Each programmable processor architecture has its own machine language for program development, but for an application developer, programming in it is most of the time overly tedious and error prone. In such a language, the programmer needs to specify almost everything explicitly. This would include deciding which instruction to execute and when (instruction selection and scheduling), also which memory or register address to store each data (data assignment). For custom architectures with SIMD instructions, the programmer would also be tasked with grouping operations on scalars into vector operations. This entails handling the data assignment and access for the vector inputs and outputs of these operations.

Traditionally, instead of the machine language, the programmer uses a higher level language (such as the C programming language) that lets her/him focus on how things are to be done, rather than the details of the architecture. The mediator here is the *compiler*. As depicted in Figure 1, the compiler takes the code written by the programmer in the high-level language (i.e. the source code) as input and outputs a translation to the target architecture's machine language (i.e. the machine code). The resulting machine code has all the details necessary to make it executable on the target architecture, as mentioned earlier. This process can be divided into three major parts: *front end*, *optimizer* and *back end* (also referred to as *code generation*) [4]. The front end of a compiler textually analyzes the source code to check its validity, both syntactic and semantic, and translates it into an *intermediate representation* (IR). The resulting IR is a data structure, generally some type of graph, that represents the source code in a way that enables further processing by the optimizer and the back end. The optimizer is responsible for platform independent optimizations over the IR, such as removing unreachable code (dead-code elimination) and avoid recomputation of expressions(common subexpression elimination). Finally, the back end is responsible for machine code generation from the IR the optimizer outputs, together with architecture specific optimizations [4, 5].

Code generation itself can be divided into three steps (subproblems), namely: instruction selection, instruction scheduling and register/data allocation. Traditionally, these steps are executed sequentially and in isolation [4]: instructions to implement the application are selected, selected instructions are scheduled, and finally, data assignment for the inputs and outputs of the scheduled instructions is decided. While it is possible and sometimes beneficial to change the order of these steps (e.g. just in time compilation of media processing applications for very large instruction word processors [6]), traditional compiler technique uses the given order of execution. Each step is hard (NP-complete) to solve optimally [4, 7]. To reach solutions in a reasonable amount of time, each step is commonly solved seperately using heuristic or approximate algorithms that generate suboptimal so-

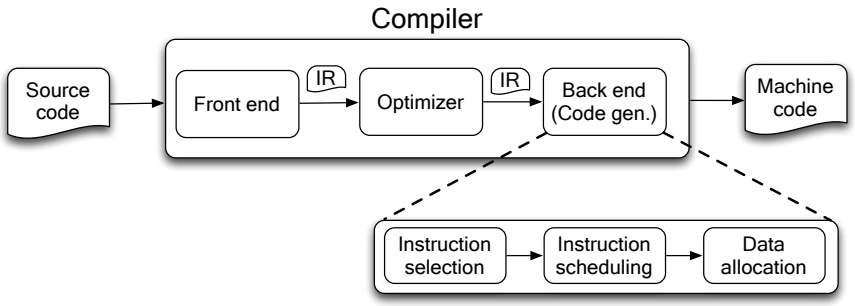


Figure 1: Structure and context of a traditional compiler [7]

lutions. For custom architectures, this is further complicated by the irregularities in the architecture [1]. Traditional techniques for each subproblem becomes harder to apply to custom architectures as these techniques are not designed to exploit the special hardware design. Moreover, the interdependence between subproblems becomes more significant for the overall result. For example, a specific vectorization of operations in scheduling may cause latency penalties because of irregular data access, depending on data allocation. In some scenarios this may offset the speedup from vectorization altogether. We experienced such a scenario in our initial experiments, where we separated the subproblems as traditional compilers do. The code generated this way had three times lower throughput than machine code written by the architect. Because of the custom nature of the architectures and the separation of subproblems, using standard techniques and tools to compile from a high level language comes at the price of very low quality of the generated machine code.

The preferred alternative for custom architectures is to write machine code by hand. But as mentioned earlier, this is a very time consuming, tedious and error prone process, as the programmer has to do the compiler's job. Furthermore, the programmer has to write machine code that uses the capabilities provided by the custom architecture, otherwise the architecture is utilized poorly, which beats the purpose of having a custom architecture. As there is no assistance from a compiler, the programmer needs to know the intricate details and complexities of architecture, including, but not limited to, processor structure, memory layout, machine instructions. Most of the time this information is available only to the architect of the processor, leaving programming solely to the architect.

In short, custom architectures are a good solution for the high-performance, low power budget demands in embedded systems, but their custom nature introduces a new challenge, which we call the *programmability bottleneck*: Traditional code generation techniques generate poor-performance code for custom architectures; therefore obtaining high-performance code is limited to programming in the machine language and requires in-depth knowledge of the architecture. In this

thesis we address the problem of programmability for custom architectures from different perspectives. The rest of the thesis is organized as follows: Chapter 2 provides a background to the papers included in the thesis, to acquaint the reader to the essential topics and concepts. Chapter 3 gives an overview of the field and related publications. Chapter 4 presents our problem formulation and Chapter 5 gives an overview of our contributions. In Chapter 6, we present our conclusions and a give glimpse into possible future work. Finally, a list of papers together with author contributions precedes the included papers themselves, in Chapter 7.

2 BACKGROUND

In this part, we briefly introduce custom architectures, code generation and constraint programming to provide a background for our work.

2.1 Custom architectures

Throughout this thesis, we targeted two custom architectures: the EIT architecture [2] and ePUMA [3]. Both of these architectures are endowed with a Single Instruction Multiple Data (SIMD) pipeline and a high-bandwidth banked memory. To familiarize the reader with these concepts, in this section we introduce pipelined execution, Single Instruction Multiple Data processing, banked memory organization and access restrictions that come with it. We conclude the section with an overview of the ePUMA architecture as an example. More details on the architectures can be found in papers II and V.

2.1.1 Pipelined execution

Exploiting the parallelism inherent to target applications is crucial to increase the throughput of an architecture. One common technique, which is also used by the architectures we target, is *pipelined execution*. Pipelined execution divides instructions into stages to run them as in an assembly line, overlapping different stages of multiple instructions. When there are enough independent instructions to run, this technique improves the utilization of the resources and increases the throughput, without changing the issue-width [4]. An example is depicted in Figure 2, where each instruction is executed in five sequential stages, each taking one clock cycle to complete. This makes the execution time of an instruction, i.e. the *latency*, 5 clock cycles. If four instructions are executed sequentially, the total execution time would be 20 clock cycles. With pipelining this number is reduced to 8 clock cycles. Moreover, after filling the pipeline in the 5th clock cycle, we get one result per cycle. This makes the *throughput* 1 instruction per cycle (IPC). Without

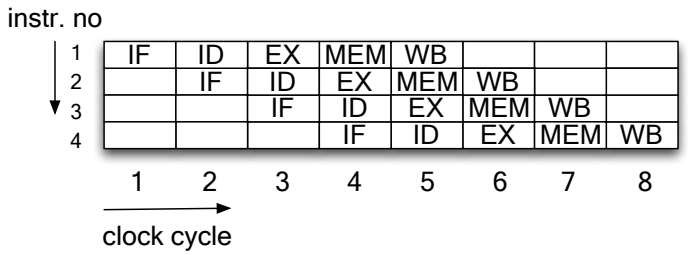


Figure 2: Pipelined execution example. Each instruction comprises five sequential parts: IF (instruction fetch), ID (instruction decode), EX (instruction execution), MEM (memory access), WB (write back).

pipelining, the throughput would be $1/5 = 0.2$ IPCs. The benefit comes from improved utilization of the hardware dedicated to each pipeline stage, by overlapping different stages from several instructions.

Instructions that are data dependent on each other may cause stalls in the pipeline, as an output in a previous instruction may be input to the following instruction. Some architectures employ techniques like *forwarding* to potentially eliminate stalls. This is done by additional hardware that can feed the result back from a pipeline stage to a previous stage for a later instruction. For the architectures we target, there is no forwarding, which means simpler hardware and simpler dependency analysis in code generation.

For architectures targeting application domains that incorporate many conflicting standards (e.g. a 4g mobile terminal complies to more than 10 different radio standards [8]), achieving flexibility together with the high-performance and low energy consumption requirements is a significant challenge. Architectures like EIT overcome this by designing dynamically reconfigurable hardware for each pipeline stage, providing a cheaper alternative to developing specific hardware for each standard, with regards to area consumption and development time [2]. In such an architecture the standards share resources and dynamically reconfigure them when necessary.

2.1.2 Single Instruction Multiple Data

As mentioned earlier, data parallelism commonly occurs in digital signal processing (DSP) and multimedia applications, where the same operation (e.g. filtering, conversion) is applied on many data points. *Single Instruction Multiple Data (SIMD)* processing units are developed to exploit this kind of special parallelism. Instead of single elements, a SIMD processor's input and output is a vector of data.

While the idea is straightforward, SIMD processors present challenges in programming and data alignment. As a general rule in parallel execution, the opera-

tions that are to be run in parallel have to be independent of each other. The SIMD paradigm adds the single instruction restriction on top of this, which makes a hard problem (identification/exposing parallelism) even harder. Organization of input and output of the SIMD unit as vectors is another challenge connected to the register and memory structure provided by the architecture. As the SIMD processor accesses vectors of data, vector accesses may be challenging as well, depending on the underlying register and memory structure,

2.1.3 Multi-bank memory and access restrictions

A vector processing unit without a memory that provides enough bandwidth to read and write vectors is useless, as the computation bottleneck will be replaced by a data/memory access bottleneck. Therefore, the memory organization of the architectures we target is an essential part of their custom nature.

The common technique to achieve high-bandwidth access for the architectures we target is to have a multi-bank memory structure. A memory is divided into banks with independent access ports. Each address in a bank contains data that can be part of a vector. Access to several banks simultaneously enables reading/writing an entire vector. This structure enables accessing each bank independently (i.e. different address for each bank) and flexibly assembling a vector from these accesses.

Both EIT and ePUMA incorporate a multi-bank memory structure. However, there are some differences. EIT takes the abstraction from scalars to vectors one step higher and works with matrices as vectors of vectors. To support this they provide a multi-bank matrix memory, where each address in a bank contains a vector. In ePUMA, each bank address contains a scalar instead. We simplify this in modeling by treating vectors as scalars and matrices as vectors for EIT. Another difference is in the number of access ports per bank. While EIT banks are dual-ported i.e. one read and one write per clock cycle is possible, ePUMA banks are single-ported i.e. one read or write per cycle is possible.

Both architectures allow some flexibility in assembling a vector from scalars, but differ in how they do it. To simplify memory access configuration, EIT divides the memory further into lines and groups banks into pages. The allowed access patterns are stored in access descriptor registers. ePUMA on the other hand allows different types of regular access without any penalty, and irregular access with possible latency penalty, depending on the other accesses happening in the pipeline at the same time. Further details on these restrictions and how we model them are presented in each paper targeting these architectures.

2.1.4 ePUMA architecture

To illustrate how these concepts fit in a custom architecture, we give an overview of the ePUMA architecture, as depicted in Figure 3. To the left, the entire system,

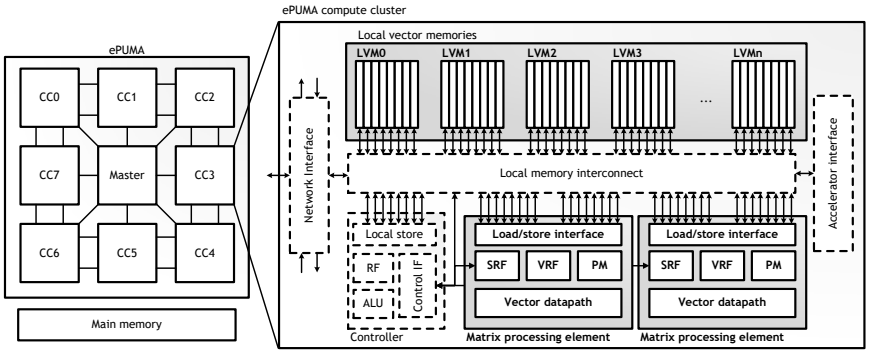


Figure 3: Overview of an example custom architecture - the ePUMA architecture.

with eight computing clusters and a master processor to control them is shown. A computing cluster contains a local controller, multiple vector DSP compute processors called Matrix Processing Elements (MPEs) and a set of local vector memories (LVMs), as shown in the right side of the figure. Each MPE can be assigned two of the local memories at a time for processing. The memories may be reassigned to exchange data between cores. When the target architecture is ePUMA, the focus of this work is limited to code generation for a single MPE, as each MPE is a complete custom architecture. The interconnection of several MPEs is the subject of another study that I contributed to [9].

A MPE has a complex pipeline structure (vector datapath), with 16 multipliers and three levels of arithmetic units, to accelerate general and application-specific computing patterns. With SIMD instructions, the processor can operate on vectors of arbitrary length directly in the local memories, processing chunks of size up to 128-bits, per clock cycle. Figure 3 also depicts the scalar register file (SRF), vector register file (VRF) and the program memory (PM) for each MPE.

The targeted version of the architecture implements instructions that have the following format:

```
op  dst src1 src2
```

where `op` is the instruction code and `dst` represents the destination of the output, while `src1` and `src2` represent the first and second operands of the instruction respectively. Commonly in signal processing, each scalar is a complex number represented by a 32-bit word. As the maximum size of a vector is 128, each vector can have 4 complex numbers. The SIMD width (SW) is configured accordingly to operate on up to 4 complex numbers at the same time. Figure 4 depicts an example instruction for the targeted version of the architecture, where $SW = 4$. The SIMD logic works point-wise over the operands and aligns operations to *lanes* 0 to $SW - 1$ of the SIMD processor.

$$\begin{array}{l} p = a + e \\ q = b + f \\ r = c + g \\ s = d + h \end{array} \quad \Rightarrow \quad \texttt{sum[p,q,r,s][a,b,c,d][e,f,g,h]}$$

Figure 4: SIMD logic

The operands in `dst`, `src1` and `src2` can reside either in the memory or a register, but each vector operand has to come from the same memory/register. Registers provide faster access while memory provides more space and flexibility. There are two 4-bank memories and 8 vector registers available. Each multi-bank memory allows reading a full vector in one clock cycle, provided that the access does not involve reading multiple elements from the same bank which constitutes a *bank conflict*. Some access patterns (henceforth called *regular access*) are supported implicitly through the hardware implementation while other patterns can be used through the help of a *permutation vector* stored in the program memory as long as it does not entail any bank conflicts.

The architecture can issue an instruction each clock cycle but instruction latency depends on several factors. Different operation types can have different latencies. We assume that the default latency for a multiplication is 4 clock cycles, i.e. the output of a multiplication is ready to use 4 cycles after its issue. The same figure for an addition is 1 clock cycle. This default latency is extended when one of the following occurs:

- *Writing back to memory:* Results in one clock cycle additional latency.
- *Bank conflict:* Results in one clock cycle additional latency per conflict, per bank.
- *Memory conflict:* Both *src*1 and *src*2 are read from the same memory. As the memory is single-ported, this adds one clock cycle latency.
- *Too many permutations:* Each irregular access needs a permutation vector to be defined and kept in the program memory. The architecture provides a way to avoid the permutation penalty if there is only one read permutation in the pipeline. But when there is more, this adds to the latency.

Registers, on the other hand, cause no such additional latency or delay penalties. However, they only allow regular access.

2.2 Code generation

Here we give an overview of code generation and briefly introduce its subproblems, as each of them is a research topic in and of itself. For more details and advanced topics that are beyond this thesis, we refer the reader to [5, 4, 6].

2.2.1 Overview

As described earlier and depicted in Figure 1, the front end of a compiler takes the source code and translates it into an intermediate representation (IR). This IR is then fed into the optimizer phase of the compiler which does code optimizations prior to code generation by the back end. In paper II we introduce a domain specific language targeting the EIT architecture, that generates an IR as the input for the code generation. Further details on IR can be found in that paper. In this section, we focus instead on code generation as it is more central to this thesis.

Code generation is the process of translating an intermediate representation of an application into machine code that can be run directly on the targeted architecture. Traditionally, code generation is divided into several subproblems in order to keep it manageable, as each subproblem in itself is very hard to solve optimally [4].

2.2.2 Instruction selection

The operations in the IR are abstract operations. They do not necessarily correspond to one instruction in the machine language. Instruction selection is the phase where the abstract operations in the IR are mapped to machine instructions. Custom architectures tend to provide complex instructions for groups of operations that are common in the target application domain. For example the ePUMA architecture provides two instructions that together implement inverse discrete cosine transformation (IDCT), a very common operation in multimedia applications. Figure 5 shows one of those instructions in a simplified IR form, where nodes represent operations and edges represent dependencies between them. Note that the instruction takes two vectors with eight scalars as input and generates one vector with eight scalars as output.

Generally the architecture enables several different implementations of the application, therefore, to select the one that performs the best becomes another objective for instruction selection. The performance metric depends on the objectives of the programmer and the application domain, but most of the time is one of minimal execution time, code size or power/energy consumption. Even though the eventual performance of the generated code is also highly dependent on the other code generation steps (i.e. instruction scheduling and data assignment), methods such as profiling are used to estimate the cost of selecting an instruction [7].

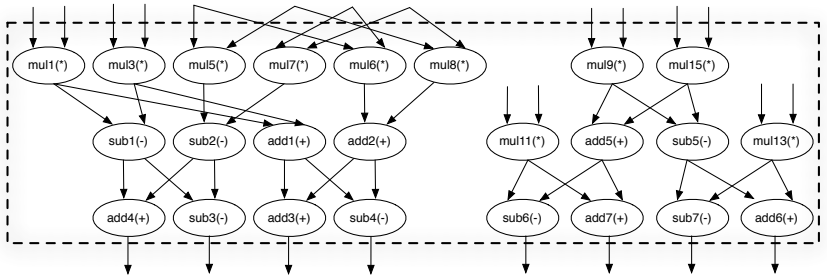


Figure 5: First half of IDCT in ePUMA, implemented by the instruction named *idct8pfw*

2.2.3 Instruction scheduling

After instruction selection, the order of execution (i.e. the schedule) for the selected instructions needs to be specified. If there are several functional units capable of executing the instructions, a schedule additionally entails the assignment of instructions to particular units. A valid schedule needs to respect the constraints of the architecture and the application. Architectural constraints include available functional units, issue width (number of instructions that can be issued simultaneously), types of operations that can be bundled together (for a SIMD unit, this is equal to one), etc. On the other hand, application constraints consist of control and data dependencies between operations.

When optimizing for high-performance i.e. high throughput, as is the case for us, the objective is to come up with a schedule that performs the best i.e. the optimal schedule, or satisfy some performance thresholds such as minimum throughput and maximum latency. To achieve this goal, it is important to exploit the parallelism inherent to the application and the instruction-level parallelism (the capability to run several instructions simultaneously) the architecture provides. Loops constitute a special source for parallelism. as independent operations from different iterations can be run in parallel. For this reason, several techniques target loops for scheduling with improved parallelism. One such technique is called *modulo scheduling* [10]. It involves finding a schedule that initiates iterations as soon as possible, taking into account dependencies and resource constraints, while also repeating regularly with a given interval (initiation interval II). *Loop unrolling* is originally a compiler optimization technique that unrolls several consecutive iterations of the loop together to decrease the loop overhead [5]. With reordering of operations, it can help eliminate stalls because of data dependences, by executing operations from a following iteration. As it possibly increases the number of independent operations, it can also be used to expose more parallelism [11]. The main downside in this case is the increase in the code size. Modulo scheduling and loop unrolling can also be combined. For further details, see paper III.

2.2.4 Data assignment

Together with the inputs and outputs of the application, the intermediate data i.e. the data produced and consumed within the application, have to reside somewhere during the interval between the define and last use time for each data (i.e. *lifetime*). Data assignment is the phase that decides on where each live data is kept.

Commonly, registers are situated closer to the processing elements and provide faster access compared to other memory units. Higher access speed comes with higher price, and therefore architectures tend to have only a limited number of registers available [12]. This entails that only a limited amount of live data can reside in registers, while the rest has to be stored in slower memory units (i.e. *spilling* to memory). Therefore, traditionally, data assignment is focused on finding a register allocation with minimal amount of spills, to minimize additional memory access latency caused by the spills. Most of the state of the art is based on a graph coloring method by Chaitin [13]. The graph coloring problem is to minimize or limit the number of colors necessary to color each node in a graph, where adjacent nodes have to be colored differently. This is a problem that predates computers, and therefore has many existing solution techniques. In order to benefit from these techniques, Chaitin uses an interference graph to represent the competition among data to be placed in a register. In this undirected graph, each node represents a data (i.e. a *temporary*) and two nodes are connected if their lifetimes overlap. With this graph, the register allocation problem is turned into a graph coloring problem.

The architectures we target are built to run data intensive applications. Therefore, they are highly dependent on memory-bandwidth, in order to achieve the required throughput. As mentioned earlier, this is addressed by customized memory structures. However, in order to achieve high bandwidth, data should be assigned and accessed in specific ways, defined by the memory structure. Otherwise, either the instruction schedule becomes invalid as some of the inputs or outputs can not be accessed, or significant latency penalties occur for conflicting accesses. As a result of the complexity in the memory structure, memory access and allocation becomes the focus of data assignment, instead of register allocation.

2.3 Constraint programming

In all the included papers we use constraint programming (CP) to model our problems. Therefore, each paper has an introduction to CP, highlighting the aspects relevant to that paper. Here, we give a more general introduction.A thorough description can be found in the Handbook of Constraint Programming [14].

The models in the CP paradigm are defined as constraint satisfaction problems over a series of variables. The variables represent the decisions that constitute a solution, such as start times of operations, memory locations and lifetimes of data. Variables are defined by the values they can take, namely their domains. A solution is an assignment of a singular value to each variable. Constraints cap-

ture the relation between variables, such as precedence between two operations, non-overlapping lifetimes for data on same location. These relations restrict the combinations of values the variables can simultaneously take.

Each constraint is paired with a consistency method (a propagator) to eliminate the infeasible values (a.k.a. *pruning*). These methods can be complete (pruning all infeasible values at once) or incomplete (pruning a subset of infeasible values), depending on the choice of algorithms implementing them. Incomplete methods are preferred when complete methods have too high algorithmic complexity. A constraint and its consistency method are often used interchangeably, i.e. "constraint" referring to the method that prunes infeasible values. Constraints are independent of each other but affect each other through the domains of the variables. This independence provides a significant amount of flexibility as constraints can be plugged in and out easily, simplifying the update and maintenance of models.

A *constraint solver* is a framework that provides the programmer with a library of built-in constraints, together with a constraint solving engine. This engine is responsible for coordination of the propagators and the guessing method e.g. search with backtracking. In a simplified view, the engine runs each propagator until a *fixed-point* (where no more pruning is possible via propagators) is reached (i.e. a round of propagation). If the reached fixed-point is not a solution, the solver needs to resort to guessing. A guess involves picking a variable and constraining its domain e.g. assigning it to a single value. This new domain can make some of the constraints invalid for some values, triggering new propagations. As a guess may be wrong, a way to backtrack from it is necessary. This is achieved by keeping track of the guesses as a tree (i.e. *the search tree*). The strategy for picking which variable to guess on (i.e the variable selection heuristic) and the strategy for constraining the selected variable's domain (i.e. value selection heuristic) decides the shape of the search tree. The search tree can be used to turn a satisfaction problem into an optimization problem. An efficient technique to search for optimality using the search tree is branch-and-bound technique [15].

A distinct feature and strength of CP is the concept of *global constraints*. A global constraint logically 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 [16]. Propagators to these constraints are commonly implemented by existing algorithms from well-studied fields such as graph theory and operations research.

Throughout this thesis, we used the JaCoP framework as our constraint solver [17]. JaCoP provides a wide selection of built-in global constraints, some of which are specially designed for scheduling (`cumulative`), and others that can be used to formulate data assignment `diff2` and access constraints `regular`.

To make things more concrete, consider the example IR in Figure 6. The graph consists of two operations and four data. There are two inputs (a and b), one output (z) and one intermediate data (x). Data b is input to both the multiplication and the

addition. The other input of the addition is the result of the multiplication, i.e. x. Therefore there is a data dependency between the operations, which is translated into a precedence constraint in the model as follows:

$$t_* + l_* \leq t_+$$

Here, t denotes the start time of the operation and l denotes its latency. For an operation, latency corresponds to the time that needs to elapse after the start of execution, for its result to be ready.

The dashed lines in the figure denote the definition point and last use of each data. If the inputs (a and b) are assumed to reside in registers or memories before the execution starts, their definition time can be assumed to be zero. However, the definition time of x depends on the multiplication, more specifically it is $t_* + l_*$, as it is defined when the result of the multiplication is ready. The last use time on the other hand, depends on the operation that finishes using the data. We assume that an operation uses an input data during its execution time, which we denote with d. Therefore, the last use time for a is $t_* + d_*$; for b and x this is $t_+ + d_+$. Note that the start times of operations (t) are variables, and will be set by scheduling decisions made by the solver. Another detail to note is that the latency (l) and the execution time (d) of an operation can be different.

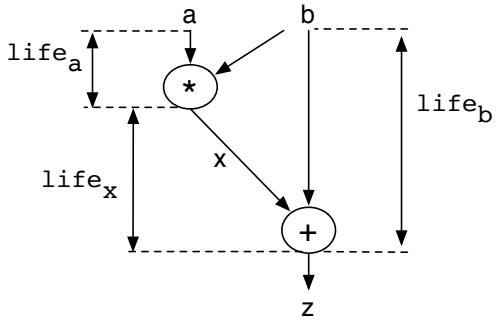


Figure 6: A simple IR. Dashed lines denote the definition point and the last use point for each data.

The time between the definition and last use identifies the *lifetime* of a data. Lifetime analysis is important in order to reuse registers and memory addresses without assigning two or more live data to the same address. In the example above, lifetimes of a and x are not overlapping, therefore they can reside in the same location. However, the lifetime of b overlaps both with a and x, therefore it can not share address with them. An overlap, therefore, happens in a two-dimensional space, the dimensions being the *address* of a data and its *lifetime*. Assuming that the size of a data does not change depending on where it is located, the assignment for each data i can be represented in this two dimensional space, as a rectangle

originating from $(address_i, def_i)$ with width $life_i$ and unit height. In this case, def_i denotes the definition time of data i. With this reasoning, data assignment can be modeled as non-overlapping rectangles, using the `diff2` global constraint. As the lifetime of a data is dependent on the start times of the operations that use or define it, the `diff2` constraint will interact with the scheduling constraints such as precedence constraints and `cumulative`.

3 Related Work

Each paper included in this thesis has a separate section on related work specific to that paper. Here, we present the related work more general to the field. Note that some newer papers are also included here, that are published after the publication of our papers. We contrast subproblem-specific works (i.e. instruction scheduling and register allocation) to more unified/integrated approaches and situate our work in the latter. Note that we do not allocate a separate section for instruction selection as we mention the most related work under unified approaches. For further reading we refer to the extensive survey on instruction selection by Blindell [7].

3.1 Instruction scheduling

Optimal instruction scheduling is a very hard problem, for a single-issue processor it is NP-complete if there is no fixed bound on the maximum latency [18]. For this reason, it is common practice to use list scheduling with a priority heuristic, instead of an exact method aiming for an optimal schedule. A compelling case for using exact techniques such as constraint programming for instruction scheduling is made in [19]. The authors performed an extensive computational study of heuristic and exact techniques for superblock instruction scheduling using realistic architectural models of processors that were commonly used in telecommunications and DSP, at the time of the study. One important conclusion of this study is that the exact scheduler (which internally uses constraint programming) always results in better code however with the downside of longer scheduling time compared to heuristic schedulers. Hence the exact methods are suitable for aggressive optimization but maybe not for general purpose compilation.

Malik et al. [20] present a superblock instruction scheduler based on CP, targeting multiple issue pipeline processors with several functional units for instructions such as load/store, integer, floating point and branch instructions. Isolating the instruction scheduling problem, they focus on the DAG representation of the superblock and make use of graph transformations presented in [21] to im-

plement implied dominance constraints. These help reduce the search space for solutions, therefore decrease optimization time. Our experiments showed that this propagation is already done when we use the `cumulative` global constraint. This makes us believe that the implementation of `cumulative` in JaCoP already covers the implied dominance constraints. The study reports optimal solutions to superblocks containing up to 2600 instructions from SPEC2000 integer and floating point benchmarks. As mentioned, the study solves instruction scheduling in isolation, while we combine it with other subproblems of code generation, i.e. instruction selection and register/memory allocation.

Modulo scheduling is a recurring method in embedded systems for loops/kernel scheduling. Many studies are reported during recent years. Kudlur and Mahlke present modulo scheduling for stream graphs in [22]. They integrate actor fission and processor assignment as an ILP as a first phase, followed by a phase that assigns actors to pipeline stages, overlapping communication and computation, to increase overall utilization. Mei et al. introduce modulo scheduling to Coarse-grained reconfigurable architectures (CGRAs) to exploit loop-level parallelism [23], while Kim et al. [24] tackle the problem of long compile times for modulo scheduling for CGRAs, caused by the difficulty of finding a good routing of operands through the processors. They propose patternized routes in order to simplify the problem, and this trade-off results in 6000 times faster compilation while preserving 70% throughput on average compared to the state-of-the-art. We on the other hand use modulo scheduling as a set of additional constraints rather than the core method for scheduling.

3.2 Register Allocation

Register allocation is a research area in and of itself. This subproblem is studied in depth for compilers targeting traditional processor architectures and most works base themselves on the seminal graph coloring method by Chaitin [13]. This method is briefly explained in Section 2.2.4. There are many combinatorial methods targeting register allocation as well, a detailed overview for these methods can be found in [25]. Here we focus instead on the studies closely related to ours, that target register allocation in isolation. Methods similar to ours that target register allocation as part of a unified approach is covered in the following section.

The work by Domagała et al. [26] uses a two-dimensional instruction tiling approach (one dimension for intra-iteration, another for inter-iteration) to expose register reuse among several unrolled loop iterations. The focus here is to minimize register pressure and spill code to avoid the high memory latency. The optimization is modeled as a constraint satisfaction problem and solved using constraint programming. Once the tiling is decided a trivial scheduling is employed. Therefore we classify this as an isolated solution for register allocation. In contrast, at

each of our studies where we target data assignment including register allocation, we combined it with other subproblems of code generation.

In [27], You and Chen present a vector-aware register allocator, targeting GPU shader processors. The target architecture provides a combination of scalar and vector operations. They observe that in the shader programs, there are many variables that are either scalar, or comprise N scalars where N is less than the size of a vector register. Therefore, they present a framework that divides vector registers into scalar parts, and allocate each variable to these slots (i.e. *element-based register allocation*). They also incorporate register packing to avoid wasting contiguous register space. The register allocator is implemented in an in-house just-in-time compiler, therefore register allocation is done before scheduling. The experiments show improvements in register utilization and decrease in spills to the memory. As ePUMA allows scalar access and groupings that are smaller than the vector register size, we also implement element-based register allocation. We do not explicitly pack registers, but this is done implicitly through scheduling of vector operations, as we integrate register and memory allocation with instruction scheduling.

3.3 Unified approaches

One of the first studies aiming at a unified approach is by Kessler and Bednarski [28]. They solve the combined instruction selection and scheduling problem with a limited number of registers, optimally, using dynamic programming. Similar to our approach, they target basic blocks that are represented as directed acyclic grpahs. However, their approach is practically limited to small graphs (< 50 nodes) for finding a solution in a reasonable amount of time. They expand this approach to cover code generation for very large instruction word (VLIW) architectures [29], but the approach is still applicable only to small graphs. In contrast, our approach can solve 4-5 times larger graphs optimally or almost optimally for combined instruction selection and scheduling.

Unison is a project aimed to combine the traditionally separated instruction selection, register/memory allocation and instruction scheduling problems into one problem to use the inter-dependency between these subproblems to achieve improved code generation [30]. For instruction selection, Blindell et al. propose a universal scheme using constraint programming in [31]. They combine control-flow with program-/data-flow to select instructions for kernels that span over several basic blocks. They incorporate a subgraph isomorphism algorithm (which is also used to implement the subgraph isomorphism propagator in JaCoP) to match instructions that are represented as pattern graphs with parts of the combined application graph. This pattern matching is similar to our approach in [32] for identifying possible instructions. It is possible to use the same idea to identify and select processor extensions as part of application compilation for reconfigurable processors as done in [33].

In [34, 35], Casteñada et al. detail the integrated register allocation and instruction scheduling for code generation, as part of Unison. Target architectures for experimental evaluation are MIPS32 and Hexagon V4, a VLIW processor included in Qualcomm's Snapdragon [36]. As we do throughout the papers that include register and memory allocation, they formulate register allocation as the non-overlapping rectangles problem. This allows them to use global constraints that capture specifically this. For bundling operations as VLIW instructions they make use of the `cumulative` global constraint as we do for both VLIW and SIMD groupings in scheduling. They also solve many subproblems of register allocation such as coalescing and register packing that we do not consider. In contrast to this focus on register allocation, we focus on the subproblems of data assignment such as permutation vector optimization and simultaneous multi-bank access problems caused by custom memories that are designed to feed the SIMD processors we have targeted. These subproblems are directly influenced by the instruction scheduling, therefore our models have full integration of memory allocation and instruction scheduling through data access constraints, while their approach integrates them only through live ranges of program variables.

Another integrated method is by Eriksson and Kessler [37], where the authors present an integer linear programming model for optimal modulo scheduling, that solves instruction selection, register allocation, instruction scheduling and instruction allocation together. Instruction selection uses a pattern matching scheme similar to our approach. They compare this integrated model to modulo scheduling with separated stages. Target architecture is a clustered VLIW, with access to a limited number of registers. They report optimal solutions for graphs with size up to 142 nodes with a time-out of 15 minutes. The differences to note compared to our work are on the target architecture and the method used. While we modeled VLIW architectures as well, our main focus was on SIMD processors and their custom memory/register structures. This entails that our work is not limited to register allocation but also allocates memory as well. While our approaches are similar, we use the constraint programming paradigm versus integer-linear programming. This gives us ease and flexibility in modeling. Comparing target applications and results, we target application graphs with size up to 250 nodes, and solve them with a time-out of 10 minutes.

3.4 SIMD specific approaches

An integrated approach that targets SIMD processors is presented in [38]. The authors extract *vectorizable codelets* from loops that enable polyhedral transformations. They model the scheduling problem as an integer linear program and incorporate polyhedral compilation framework to extract scheduling constraints. Our work could also be extended to include constraints derived from polyhedral transformations. On the other hand, as mentioned earlier, the architectures we tar-

get come with custom memory and register organizations that need to be taken into consideration during scheduling. This complicates the vectorization significantly as each vector access to the memory is subject to restrictions which may result in delay penalties, which is addressed in our work.

Kim and Han [39] focus on SIMD code generation for irregular kernels with array indirection, which makes auto-vectorization a difficult task. Working on data-flow graphs, they exploit both intra- and inter-iteration parallelism for loops. Inter-iteration parallelism is covered by superword level parallelism, in which the source and result operands of a SIMD operation are packed in a storage location [40]. For intra-iteration parallelism, they identify the vectorizable operations within the loop. They account for the overhead in data reorganization operations such as load, store and shifting, and optimize placement of necessary data reorganization code. Our approach works as well for irregular loops, making it possible to generate efficient code regardless if the original code has regular or irregular references. We make use of the custom nature of the architecture and try to minimize data permutations without employing data reorganization code.

Another approach by Hormati et al. [41] takes the SIMD code generation for streaming applications to a higher level and focuses on *SIMDization* of actors in a streaming program. They call this macro-SIMDization. This perspective provides high-level information such as execution rates of actors and communication patterns among them, valuable for vectorization. Using their terminology, we focus in micro-SIMDization, targeting kernels that are run many times within an application. If these kernels are implemented as actors in a streaming application, macro-SIMDization could be used to vectorize the operations that can not be vectorized with our methods because of data dependences.

4 PROBLEM STATEMENT

As introduced earlier, custom architectures come hand in hand with a programmability bottleneck. One way to overcome the programmability bottleneck is through an automatic code generator that provides high quality machine code, that is competitive with machine code written by the architect. On the other hand, custom architectures are becoming more and more common, and each architecture has its own custom capabilities and restrictions. Even these capabilities and restrictions are open to change as the these architectures tend to version frequently. Therefore it is important to devise a strategy that enables building a code generator that can be changed, customized and maintained easily. Otherwise, for each version or new architecture, a new code generator needs to be built from scratch, wasting many man-hours.

This thesis is the result of our effort to overcome the programmability bottleneck of custom architectures by automating the code generation process. Two major goals for a code generation framework targeting custom architectures are:

- The machine code it generates performs at least as well as machine code written by the architect.
- It is easy to adapt it to different architectures or versions, without requiring the development of specific solutions from scratch for each new target.

Such a code generator should also avoid the shortcomings of code generation in traditional compilers for custom architectures. These include:

- Poor utilization of the special hardware.
- Neglecting the interdependence of subproblems by staging them.
- Difficult to adapt to architectural change.

To avoid these shortcomings, we propose code generation frameworks modeled using the *constraint programming* (CP) paradigm. CP fits the irregular nature

of custom architectures, as constraints are designed to be defined and to work independently. Once a skeleton model is built to capture essential parts of code generation, special hardware capabilities and requirements, or switching the target architecture can be reflected by plugging constraints in or out. This way, using CP can address the poor utilization of special hardware and achieve the goal of easy adaptation to architectural change. To reflect the interdependence of the subproblems of code generation, we aim for unified models that combine the targeted subproblems.

As our main focus is on code generation, three out of the five included papers directly target its subproblems. Paper I presents a combination of instruction selection and scheduling of complex instructions for DSP kernels. Paper II combines instruction scheduling and memory allocation for a dynamically reconfigurable custom vector architecture (EIT). Finally, paper V combines instruction scheduling, data allocation (both memory and register) and data access patterns in a single model, targeting another SIMD architecture (ePUMA). The remaining two papers deal with topics relatively peripheral to code generation. Paper III is a comparative study on scheduling techniques for kernels running on architectures with SIMD pipelines. Paper IV presents a design space exploration framework for assisting architectural decisions on custom vector architectures.

5 OVERVIEW OF CONTRIBUTIONS

- *A high-level programming framework for custom architectures with SIMD capabilities and complex memory organization.*

  We propose a framework that can take a dependency graph as intermediate representation, that is generated from code written in a high-level language, and generate high quality machine code for the target architecture automatically. As the high-level language for our purpose, we developed an in-house domain specific language, while a dependency graph generated from another high-level language would work as well. The framework enables instruction scheduling with SIMD groupings and data allocation with optimized data access patterns. For further details, see paper V.

- *Formulation of modulo scheduling as part of a constraint programming model for both code generation and design space exploration.*

  While it is commonly used separately, we integrated modulo scheduling into a code generation framework and design space exploration. The integration uses a novel constraint-based formulation of the problem and does not require major changes in the rest of the constraint model. This integration is part of papers II, III and IV.

- *An automata based formalization of the restrictions on data access patterns for a custom memory.*

  The custom, multi-bank memory that comes with the SIMD processing unit for one of the architectures we target, enables vector access with or without additional latency, depending on the access pattern. For our constraint model, we formalized this as an automaton, using the `regular` global constraint. The only similar attempt we found in the existing literature [42], focuses on minimizing cache misses in program-level, and targets scalar

processing. Our contribution is on minimizing the latency caused by irregular vector accesses, happens at the instruction-level, and targets vector processing. Details can be found in paper V.

- *A method for fast exploration of application specific architectures.*

  The architectures we targeted are constantly under improvement and change, based on the application requirements. Keeping this in mind, we developed a method for exploring potential architectural configurations for application set specific SIMD processor architectures. We employed constraint programming, and formulated the problem as Pareto optimization in a three dimensional space (number and width of SIMD units, number of scalar units), using modulo scheduling to ensure throughput requirements are met. This contribution is the main focus of paper IV.

- *Formalization and integration of subproblems of code generation as a constraint satisfaction problem.*

  To reflect the fact that the subproblems of code generation are intertwined, we merged the subproblems we targeted, and formalized them as a unified constraint satisfaction problem. This is in contrast to the traditional compiler technique of solving each step separately and merging the solutions.

  In paper I, we integrate instruction selection and scheduling. Paper II combines instruction scheduling with memory allocation. Finally, paper V integrates instruction scheduling and data assignment (both register allocation and memory allocation) with optimized data access patterns. Note that the idea of a unified model is not novel in itself, while the integration of data access constraints into a unified model is, to the best of my knowledge.

6 CONCLUSIONS

6.1 Summary

Custom architectures are powerful tools for achieving high performance with low cost. However, the fact that they are extremely hard to program limits their use to a handful of programmers and therefore limits their benefits. Our work is a step towards alleviating this problem by making these architectures easier to program. Papers I, II and V demonstrate how we successfully address subproblems of code generation for custom architectures in a more unified manner compared to traditional compilers. In our experiments we compare the schedules and machine code we generated to theoretical lower bounds and manual code written by experts in the field and of the target architecture. We targeted kernels from real-life applications that are common for the target architecture, with varying IR sizes and shapes. For these applications, we either matched or got close enough to either the theoretical bound or the manual code, targeting two different custom architectures. Therefore, we deem that our automatic code generation scheme achieves the goals we laid out in the problem statement.

Using constraint programming, we built flexible yet highly detailed models of the target architectures and applications. All the models in this thesis share the same skeleton for the overlapping problems, such as instruction scheduling. The rest of the model is extended from this skeleton by plugging in new variables and constraints. This flexibility of modeling in constraint programming allows one to experiment with the level of abstraction when modeling architectures. Adding or removing a detail in the architecture corresponds to adding or removing a group of variables and constraints. A good example of this flexibility is the models for different scheduling techniques in paper III, where the shared "skeleton" corresponds to 70%-80% of the models.

During our discussions with the architects of the systems we targeted, we realized that these architectures are under constant development and update, based on the application requirements. Therefore, additionally to code generation, we

proposed a preliminary framework for design space exploration of custom architectures, with focus on meeting the requirements of the target application domain. Here, the exploration parameters are limited to the properties of the SIMD processing unit and the number of scalar units. For a more comprehensive design space exploration, more parameters should be taken into account, such as the number of registers and properties of the memory.

6.2 Future work

The problems we target are inherently hard. Both instruction selection and instruction scheduling are proven to be NP complete. Even though constraint programming provides flexibility in modeling, this algorithmic complexity forced our models to be very complex in turn, with many different types of variables and complicated constraints and complex search heuristics. The combination of a hard problem and a complex model can turn a constraint solver into a black box. Problems with reasonable sizes get quickly untraceable. In a future work, a mathematical analysis of what kinds of graphs we can solve in a reasonable amount of time should be provided for sound reasoning, instead of relying only on experiments.

For the entirety of the thesis we limited ourselves to the EIT and ePUMA architectures, together with some theoretical architecture models (such as a generic very large instruction word processor). However, it is important to target other custom architectures in order to ensure the robustness of our technique. It would be interesting to target custom architectures that provide some sort of programming support (other than assembly) such as [43], and compare the code our tools generate with the code generated by the existing tools.

Our experiments showed that for applications larger than a certain size (corresponding to 256 nodes in the IR) our techniques do not terminate with a solution in a reasonable amount of time. In order to address this, we would like to investigate graph partitioning methods similar to the ones presented in [33]. There are many aspects to this problem, as a graph could be partitioned "vertically" or "horizontally", in an overlapping or independent fashion, with different partition sizes, and with different order of partitions to solve. The problem gets further complicated with constraints on data allocation and access, as a decision in one partition may render another partition infeasible.

From a constraint programming perspective, we quite often ended up with constraints that worked orthogonally to each other, but were actually interdependent in the bigger picture. An example is the interdependency of data access constraints and scheduling constraints in paper V. It would be interesting to see if it is beneficial to develop a set of global constraints that act as a combination of these orthogonal constraints. Later on, these constraints could be generalized for other purposes.

We suspect that symmetrical solutions comprise a big part of the search space in our problems. Eliminating these symmetries would result in a smaller and hopefully more manageable search space. Another way to shrink the search space is to decide whether or not the reached state in a search node (with all the variable domains and the constraints) is subsumed by another node. This is referred to as dominance and can limit the number of search nodes to visit significantly [44].

As mentioned earlier, we used the JaCoP framework as our constraint solver, as it provides a rich set of built-in constraints. It is also built as an open source library in Java, which makes it easier to program with and merge with additional code such as pre- and post-processing and supporting data structures, if necessary. However, there are many other constraint programming frameworks available. Some of these are particularly interesting as they focus on advanced search techniques such as lazy-clause generation [45], which is shown to be particularly useful in solving resource-constrained scheduling problems [46]. In future work, we would like to investigate whether or not another solver suits our problem better.

7 LIST OF PAPERS

7.1 Papers included in the thesis

For all the papers, all authors contributed to the writing. Additional to writing, my supervisors Prof. Krzysztof Kuchcinski and Dr. Flavius Gruian contributed with discussions during each study and editing the papers.

1. *Instruction Selection and Scheduling for DSP Kernels*

   Mehmet Ali Arslan and Krzysztof Kuchcinski, Microprocessors and Microsystems, Volume 38, Issue 8, (pp. 803-813), 2014

   Contributions: I reformulated Prof. Kuchcinski's previous idea on using pattern matching for identifying instruction extensions as a method for instruction selection and implemented the unified model for instruction selection and scheduling for the targeted architectures, with constraint programming, together with the experimentation.

2. *Programming Support for Reconfigurable Custom Vector Architectures*

   Mehmet Ali Arslan, Krzysztof Kuchcinski, Flavius Gruian and Yangxurui Liu, PMAM 2015: The 6th International Workshop on Programming Models and Applications for Multicores and Manycores, February 07 - 11, San Francisco, CA, USA, 2015

   Contributions: I developed the domain specific language (DSL), the instruction scheduler and register allocator for the custom vector processor architecture developed at the EIT department in Lund university. Yangxurui Liu implemented the applications used for experimentation in the DSL.

3. *A Comparative Study of Scheduling Techniques for Multimedia Applications on SIMD Pipelines*

   Mehmet Ali Arslan, Flavius Gruian and Krzysztof Kuchcinski, DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS 2015), March 03, Grenoble, France, 2015

Contributions: Dr. Gruian proposed to carry out a comparative investigation of scheduling techniques that were of interest to us. I formalized and implemented the separate constraint programming model for each scheduling technique and carried out the experimentation and further analysis.

4. *Application-Set Driven Exploration for Custom Processor Architectures*

   Mehmet Ali Arslan, Flavius Gruian and Krzysztof Kuchcinski, 26th IEEE International Conference on Application-specific Systems, Architectures and Processors, July 27-29, Toronto, Canada, 2015

   Contributions: Based on paper 3, after discussing with Dr. Gruian, we came up with the idea for a design space exploration scheme which I designed and implemented, together with the case study.

5. *Code Generation for a SIMD Architecture with Custom Memory Architecture*

   Mehmet Ali Arslan, Flavius Gruian, Krzysztof Kuchcinski and Andréas Karlsson, Conference on Design & Architectures for Signal & Image Processing October 12-14, Rennes, France, 2016

   Contributions: I studied the ePUMA architecture that requires special code generation, because of its SIMD capabilities and custom memory architecture. Based on my studies I developed the code generation backend. I implemented some of the applications in the DSL, which was developed previously by me in paper 2. Andréas Karlsson implemented the rest of the applications and ran the generated assembly code in the simulator.

7.2 Papers not included in the thesis

1. *Partitioning and Mapping Dynamic Dataflow Programs*

   Mehmet Ali Arslan, Jörn W. Janneck and Krzysztof Kuchcinski, Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), 2012 (pp. 1452-1456). IEEE

2. *Mapping and Scheduling of Dataflow Graphs—A Systematic Map* Usman Mazhar Mirza, Mehmet Ali Arslan , Gustav Cedersjo, Sardar Muhammad Sulaman and Jörn W. Janneck, 48th Asilomar Conference on Signals, Systems and Computers, 2014 (pp. 1843-1847). IEEE

3. *Support for Data Parallelism in the CAL Actor Language*

   Essayas Gebrewahid , Mehmet Ali Arslan, Andréas Karlsson and Zain Ul-Abdin, In Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing (pp. 2). ACM

References

[1] Paolo Ienne and Rainer Leupers. *Customizable embedded processors: design technologies and applications*. Academic Press, 2006.

[2] Chenxin Zhang. "Dynamically Reconfigurable Architectures for Real-time Baseband Processing". PhD thesis. Lund University, 2014. URL: http://lup.lub.lu.se/record/4406448/file/4406451.pdf.

[3] Andréas Karlsson, Joar Sohl, and Dake Liu. "ePUMA: A Processor Architecture for Future DSP". In: *International Conference on Digital Signal Processing (DSP), Singapore, 21-24 July, 2015*. 2015.

[4] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[5] Alfred V. Aho and Jeffrey D. Ullman. *Principles of compiler design*. Addison-Wesley, 1977.

[6] Sid Touati and Benoit de Dinechin. *Advanced Backend Optimization - Iste*. 1st. Wiley-IEEE Press, 2014.

[7] Gabriel Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer, 2016.

[8] Fabien Clermidy, Christian Bernard, Romain Lemaire, Jérôme Martin, Ivan Miro-Panades, Yvain Thonnart, Pascal Vivet, and Norbert Wehn. "MAGALI: A Network-on-Chip based multi-core system-on-chip for MIMO 4G SDR". In: *2010 IEEE International Conference on Integrated Circuit Design and Technology*. June 2010, pp. 74–77.

[9] Essayas Gebrewahid, Mehmet Ali Arslan, Andréas Karlsson, and Zain Ul-Abdin. "Support for data parallelism in the CAL actor language". In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. ACM. 2016, p. 2.

[10] Monica Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 318–328.

[11] Jack W. Davidson and Sanjay Jinturkar. "Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation". In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. MICRO 28. Ann Arbor, Michigan, USA: IEEE Computer Society Press, 1995, pp. 125–132. URL: http://dl.acm.org/citation.cfm?id=225160.225184.

[12] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[13] Gregory J. Chaitin. "Register Allocation & Spilling via Graph Coloring". In: *SIGPLAN Not.* 17.6 (June 1982), pp. 98–101. URL: http://doi.acm.org/10.1145/872726.806984.

[14] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[15] Ailsa H. Land and Alison G. Doig. "An automatic method of solving discrete programming problems". In: *Econometrica: Journal of the Econometric Society* (1960), pp. 497–520.

[16] Willem-Jan van Hoeve and Irit Katriel. "Handbook of Constraint Programming". In: Foundations of Artificial Intelligence. Elsevier Science, 2006. Chap. Global Constraints.

[17] Krzysztof Kuchcinski and Radoslaw Szymanek. *JaCoP Library. User's Guide*. 2014. URL: `http://www.jacop.eu`.

[18] John L. Hennessy and Thomas Gross. "Postpass code optimization of pipeline constraints". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.3 (1983), pp. 422–448.

[19] Michael Chase, Abid M. Malik, Tyrel Russell, R. Wayne Oldford, and Peter van Beek. "A computational study of heuristic and exact techniques for superblock instruction scheduling". In: *Journal of Scheduling* 15.6 (2012), pp. 743–756.

[20] Abid M. Malik, Tyrel Russell, Michael Chase, and Peter van Beek. "Optimal superblock instruction scheduling for multiple-issue processors using constraint programming". In: *School of Computer Science, University of Waterloo, Technical Report CS-2006-37* (2006).

[21] Mark Heffernan and Kent Wilken. "Data-dependency graph transformations for instruction scheduling". In: *Journal of Scheduling* 8.5 (2005), pp. 427–451.

[22] Manjunath Kudlur and Scott A. Mahlke. "Orchestrating the execution of stream programs on multicore platforms". In: *ACM SIGPLAN Notices*. Vol. 43. 6. ACM. 2008, pp. 114–124.

[23] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling". In: *Computers and Digital Techniques, IEE Proceedings-*. Vol. 150. 5. IET. 2003, pp. 255–61.

[24] Wonsub Kim, Yoonseo Choi, and Haewoo Park. "Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.4 (2013), p. 58.

[25] Roberto Castañeda Lozano and Christian Schulte. "Survey on combinatorial register allocation and instruction scheduling". In: *arXiv preprint arXiv:1409.7628* (2014).

[26] Lukasz Domagała, Duco van Amstel, Fabrice Rastello, and Ponuswamy Sadayappan. "Register Allocation and Promotion Through Combined Instruction Scheduling and Loop Unrolling". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: ACM, 2016, pp. 143–151.

[27] Yi-Ping You and Szu-Chien Chen. "VecRA: A Vector-Aware Register Allocator for GPU Shader Processors". In: *ACM Trans. Embed. Comput. Syst.* 15.4 (Sept. 2016), 64:1–64:30. URL: http://doi.acm.org/10.1145/2961026.

[28] Christoph Keßler and Andrzej Bednarski. "A dynamic programming approach to optimal integrated code generation". In: *ACM SIGPLAN Notices*. Vol. 36. 8. ACM. 2001, pp. 165–174.

[29] Christoph Kessler and Andrzej Bednarski. "Optimal integrated code generation for VLIW architectures". In: *Concurrency and Computation: Practice and Experience* 18.11 (2006), pp. 1353–1390.

[30] *Unison: Robust, Scalable, and Open Code Generation by Combinatorial Problem Solving*. URL: http://www.gecode.org/~schulte/projects/unison.html.

[31] Gabriel Hjort Blindell, Roberto Castañeda Lozano, Mats Carlsson, and Christian Schulte. "Modeling Universal Instruction Selection". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 609–626.

[32] Mehmet Ali Arslan and Krzysztof Kuchcinski. "Instruction selection and scheduling for DSP kernels". In: *Microprocessors and Microsystems* 38.8 (2014), pp. 803–813.

[33] Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch, and François Charot. "Constraint programming approach to reconfigurable processor extension generation and application compilation". In: *ACM transactions on Reconfigurable Technology and Systems (TRETS)* 5.2 (2012), p. 10.

[34] Robert Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. "Combinatorial Spill Code Optimization and Ultimate Coalescing". In: *ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES' 14. Edinburgh, UK, June 2014.

[35] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. "Register allocation and instruction scheduling in Unison". In: *Proceedings of the 25th International Conference on Compiler Construction*. ACM. 2016, pp. 263–264.

[36] Qualcomm Technologies Inc. *Hexagon V4 Programmer's Reference Manual*. Aug. 2013.

[37] Mattias Eriksson and Christoph Kessler. “Integrated code generation for loops”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.1 (2012), p. 19.

[38] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. “When Polyhedral Transformations Meet SIMD Code Generation”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: ACM, 2013, pp. 127–138.

[39] Seonggun Kim and Hwansoo Han. “Efficient SIMD Code Generation for Irregular Kernels”. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’12. New Orleans, Louisiana, USA: ACM, 2012, pp. 55–64.

[40] Samuel Larsen and Saman Amarasinghe. *Exploiting superword level parallelism with multimedia instruction sets*. Vol. 35. 5. ACM, 2000.

[41] Amir H. Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott A. Mahlke. “MacroSS: macro-SIMDization of streaming applications”. In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 1. ACM. 2010, pp. 285–296.

[42] Jinseong Jeon, Keoncheol Shin, and Hwansoo Han. “Abstracting Access Patterns of Dynamic Memory Using Regular Expressions”. In: *ACM Trans. Archit. Code Optim.* 5.4 (Mar. 2009), 18:1–18:28. URL: `http://doi.acm.org/10.1145/1498690.1498693`.

[43] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanovi. “The Hwacha Vector-Fetch Architecture Manual, Version 3.8.” In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262* (2015).

[44] Geoffrey Chu and Peter J. Stuckey. “Dominance breaking constraints”. In: *Constraints* 20.2 (2015), pp. 155–182.

[45] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. “Propagation via lazy clause generation”. In: *Constraints* 14.3 (2009), pp. 357–391.

[46] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G Wallace. “Solving RCPSP/max by lazy clause generation”. In: *Journal of Scheduling* 16.3 (2013), pp. 273–289.

INCLUDED PAPERS

PAPER I

INSTRUCTION SELECTION AND SCHEDULING FOR DSP KERNELS

Abstract

As custom multicore architectures become more and more common for DSP applications, instruction selection and scheduling for such applications and architectures become important topics. In this paper, we explore the effects of defining the problem of finding an optimal instruction selection and scheduling as a constraint satisfaction problem (CSP). We incorporate methods based on sub-graph isomorphism and global constraints designed for scheduling. We experiment using several media applications on a custom architecture, a generic VLIW architecture and a RISC architecture, all three with several cores. Our results show that defining the problem with constraints gives flexibility in modeling, while state-of-the-art constraint solvers enable optimal solutions for large problems, hinting a new method for code generation.

1 Introduction

New demands on performance and power consumption lead to introduction of special execution platforms that are built specially to fulfill the specific requirements of a given set of applications. The platforms are usually built around a network that connects processors and application-specific instruction-set processors

Mehmet Ali Arslan and Krzysztof Kuchcinksi
Dept. of Computer Science, Lund University, 221 00 Lund, Sweden

*Based on the paper published on Microprocessors and Microsystems, Volume 38, Issue 8, Page 803, to 813, 2014.

(ASIPs). ASIPs have specific instruction sets that support a given class of applications. They also have different architectural solutions for parallel execution. Among others they usually support instructions defining complex functionalities, VLIW-like instructions and SIMD execution model. For example, the ePUMA architecture contains the Sleipnir processor [1] that has special instructions that can compute DSP kernel applications, such as DCT or IDCT, using only two processor instructions. Moreover, the processor can make use of data parallelism present in an application and execute an instruction in the SIMD mode. Other DSP processors endorse the possibility to execute several operations, such as addition, multiplication and load/store, in long instructions following the VLIW principle.

Code generation is commonly divided into three subproblems; instruction selection, instruction scheduling and register allocation. Each of these subproblems is hard to solve optimally in general and especially for realistic instances of the architectures mentioned above, as using all the mentioned features effectively is very difficult. Traditionally, to keep the problem manageable, each subproblem is solved in isolation using heuristics which trades optimality for solving time. These subsolutions are then combined to result in generated machine code. However, even if the subsolutions are optimal, dividing the problem in such a way is bound to ignore many optimization possibilities that are available when the problem is considered as a whole.

In this paper, we address instruction selection and instruction scheduling for such architectures, while incorporating their custom nature in the process as much as possible. We combine selection and scheduling in a single constraint programming (CP) model, providing the opportunity to achieve high quality solutions that are often optimal for many DSP kernels, as indicated by our experimental results. Moreover, our approach makes it possible to easily define new constraints for new architectures and define related models for instruction selection and scheduling. It also makes it possible to combine different constraints and, for example use complex instructions together with their parallel execution, achieving in this way the SIMD execution principle.

We assume that kernels written in some high-level language, such as C or CAL [2], are compiled to data-flow graphs (DFGs) [3, 4]. We only consider kernels that can be statically scheduled and do not have feedback edges. Such a DFG graph and an instruction set, defined also in DFG format, are input to our instruction selection and scheduling system. Sub-graph isomorphism is then applied to find out possible use of instructions and then instruction selection and scheduling is performed. Instruction selection and scheduling use methods offered by CP to find partially ordered instructions that implement a given kernel, resulting in the shortest schedule. Our scheduling is not limited to sequential execution and can be used both for VLIW-like processors, SIMD executions and parallel execution on several processors.

In the next section, we introduce constraint programming (CP) as used in this paper. In section 3 we discuss different approaches to instruction selection and

scheduling and compare them with our approach. Section 5 presents our formulation for the problem and the solution method. In section 5 we present experimental data obtained for different DSP kernels and discuss the results and future work in section 6. Finally, in section 7 we present the conclusions for our work.

Our previous work on this subject has been reported in [5]. Here, we extend that work by new methods and experimental results that are discussed in detail and more thorough comparison to related work.

2 Constraint Programming

In this paper we extensively use constraint satisfaction methods implemented in the constraint programming environment JaCoP [6]. We introduce constraint programming and related constraints used in this work briefly in this section.

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*, $\mathscr{D} = \{D_1 \times D_2 \times \ldots \times D_n\}$ 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.

A global constraint on the other hand, combines several simpler constraints and handle them together. is a conjunction of several simpler constraints. 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 [7]. In this paper we use intensively several global constraints, such as `Cumulative`, `Diff2` and `SubGraphMatch`.

`Cumulative` constraint [8] 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` [9] 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 2-dimensional rectangles and assures that for each pair of i, j $(i \neq j)$ of 2-dimensional rectangles, there exist at least one dimension k where i is after j or j is after i. The 2-dimensional rectangle is defined by a tuple $[O_1, O_2, L_1, L_2]$, where O_i and L_i are respectively called the origin and the length of the 2-dimensional rectangle in *i-th* dimension. The `Diff2` constraint is used in this paper for defining constraints both for resource binding and scheduling.

Finally, the graph matching constraint (`SubGraphMatch`) [10] defines conditions for (sub-)graph isomorphism between target and pattern graphs (the pattern graph can be defined as a set of separate sub-graphs). It uses a pruning algorithm developed for applications in reconfigurable computing and instruction selection.

The graph definition for this constraint defines always a vertex of a graph as a node and its ports. For example, an addition node can have two input ports and one output port while subtraction will have two input ports of different kinds to distinguish between minuend and subtrahend.

The graph matching constraint defines a number of rules specifying conditions for sub-graph isomorphism. Our matching function assigns a target node to a node of a pattern graph as it is assumed by the classical definition. The matching conditions are as follows. First, the labels of the vertex must be the same and the ports of target graph can be one-to-one mapped onto ports of the pattern graph. Second, corresponding edges between mapped nodes must exist. The algorithm developed implements these conditions as the rules that prune impossible matchings. Moreover, the filtering algorithm is only applied to recently changed variables that makes the constraint even more efficient.

A *solution to a CSP* is an assignment of a value to each variable from its domain, in such a way that all constraints are satisfied. The specific problem to be modeled will determine whether we need *just one solution*, *all solutions* or an *optimal solution* given some cost function defined in terms of the variables.

The constraint solver uses consistency methods designed for each constraint and systematic search procedures. *Consistency methods* try to remove inconsistent values from the domains in order to reach a set of pruned domains such that their combinations are valid solutions. Each time a value is removed from a FD, all the constraints that contain that variable are revised. Most consistency techniques are not complete and the solver needs to explore the remaining domains for a solution using search, which consists of systematically assigning values from variable domains to the variables. It is implemented as depth-first-search. The consistency method is called as soon as the domains of the variables for a given constraint are pruned. If a partial solution violates any of the constraints, backtracking will take place, reducing the size of the remaining search space.

3 Related Work

Instruction selection and scheduling for a given processor or multi-processor are complex problems known to be NP-complete. Special attention has been recently given to special architectures that have complex instructions and non-regular instruction sets as well as possible reconfigurability of the processor under runtime. This makes it difficult to use well known compiler infrastructures, such as LLVM [11].

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 [12] explores optimal or highly optimized code generation techniques for in-order issue superscalar processors and various VLIW processors, using dynamic programming and 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 constraint programming (CP) approach, developed during recent years for different purposes, is one of the methods used for solving these problems optimally. Beek and Wilken [13] use CP to optimally schedule basic block instructions on single-issue RISC processors. They address arbitrary latencies for instructions. This work is particularly relevant to our work since they also use the MediaBench [14] benchmark applications. An important difference though is that we consider several RISC processing units running in parallel, while they schedule the basic blocks only on a single processing unit.

The work in [15] use CP based register allocation and scheduling. They use LLVM as compiler front-end and assume that instruction selection has already been done yielding a representation of the input program in SSA (static single assignment). They perform register allocation by using `Diff2` constraints and rectangles defining define-use times for variables. On top of register allocation, they present a decomposition-based code generation technique where they locally schedule the instructions in each basic block and optimize execution cycles based on an estimated block execution frequency. The approach presented here concentrates on instruction selection and scheduling but can be combined with the methods proposed in [15] and similar methods proposed for operator based architectures in [16].

Optimal basic block instruction scheduling for multiple-issue processors by Malik et al. [17] has been an influential work. Using constraint programming, they schedule basic blocks from the SPEC 2000 integer and floating point benchmarks. The architectural model is very similar to our VLIW-like model, where several processing units run different types of basic instructions. Similar to our model, applications are represented as DAGs. The main difference to our work is the structure of the instructions. While an instruction is an atomic operation in their model, we support instructions comprising several atomic operations.

Our previous work used CP for instruction identification, selection and scheduling for reconfigurable processor extensions [10]. It uses sub-graph isomorphism for instruction identification and selection as well as CP-based scheduling. In the current work we assume a given instruction set but we extend our previous approach by addressing new architectures. In particular, we consider very complex instructions, VLIW processors as well as multicore RISC's. Previously only RISC

processors with an accelerator [10] or operator based reconfigurable architectures [16] were considered.

4 Our Approach

In this section we detail our CSP formulation together with the inputs to the system and the assumptions enclosing it. Since our model addresses a variety of problems step by step, rest of the section is divided to explain our approach in a similar fashion. First, we generate all possible instruction matches on the application graph (section 4.2). Once the matches are generated, we introduce the constraints that select and schedule a subset of them that cover the application graph completely, taking architecture particularities into account (sec 4.3). The selected instructions must be assigned to an available computational unit as well (section 4.4). Finally, as most consistency techniques are not complete, we define how the constraint solver plows through the search space (section 4.5).

4.1 Inputs and Assumptions

The input to our system is an application that is compiled to a DFG together with the DFGs of available instructions. The aims are to select the instructions that can implement the entire application and find a shortest schedule with the selected instructions.

We consider three architectures with different granularity for instructions: Sleipnir, VLIW and RISC.

The sleipnir architecture [1] can run special instructions for computing DSP kernel applications. One such instruction that comprises 24 operations is depicted in Fig. 1. In this work we consider Loeffler's algorithm for IDCT [18] on Sleipnir, which takes only two instructions to complete IDCT with eight words and eight constants as input. We also handle several Sleipnir processing units running in parallel.

For representing VLIW architectures, we use a generic model where an instruction comprises a set of operations. Throughout this section, we use the specific model where an instruction comprises only basic operations i.e. one add or subtraction; one multiplication and one load or store operation. For RISC architectures, our model consists of several cores running in parallel, where each core can run one single operation (add, subtract, multiply, load, store) at a time.

We will use the application and instruction graphs depicted in Fig. 2 as a running example in the rest of this paper. The figure depicts an application with two multiplications and four additions. Each operation is represented by a node with an identification label, comprising the type of the operation and an identity number (referred to as node id in the following). The instructions are also depicted in the same format, although without the identification labels. Instruction two and

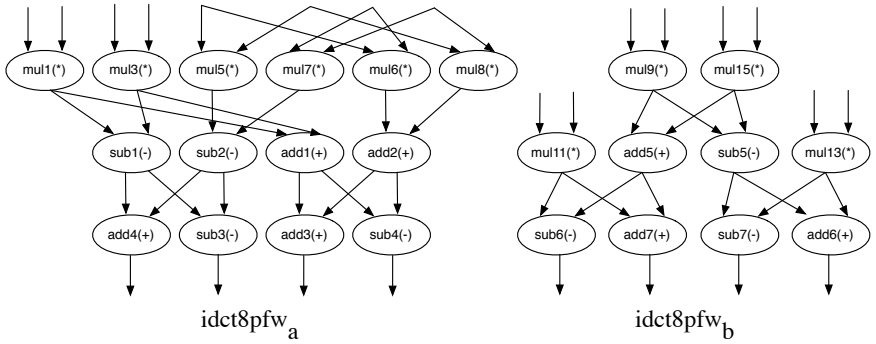


Figure 1: *idct8pfw* Sleipnir instruction that covers the half of Loeffler's IDCT algorithm [18]

three are one-operation instructions while instruction one comprises a multiplication followed by an addition.

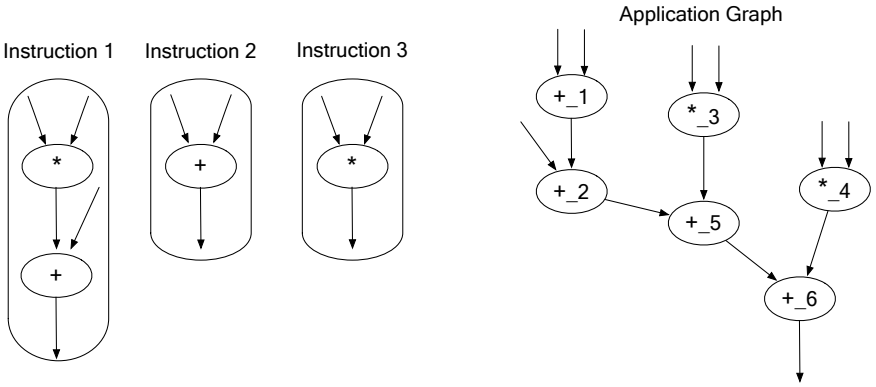


Figure 2: Example with an application graph and several instruction graphs

In this simple example, we assume that there is only one processing unit that can run all three instructions, but only one at a time. Each instruction is assumed to take one cycle to execute (duration $\triangle t = 1$), while in the general case different instructions can have different execution times.

4.2 Instruction Matching

Each instruction can be present in several sub-graphs of the application graph (i.e. sub-graph isomorphic to the application graph) as shown in Fig. 3 and 4. We call each of these sub-graphs a *match*. The problem is to select the matches that give a

full cover and shortest schedule. Since each match needs an identity (so that they can be referred to later on), we number them as illustrated in the Fig. 3 and 4.

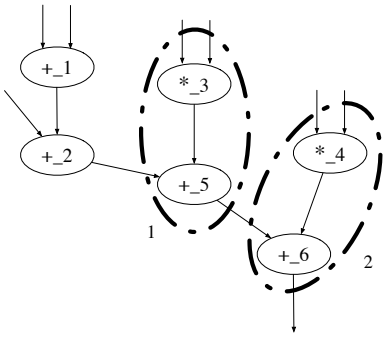


Figure 3: Matches for instruction 1, shown in dashes.

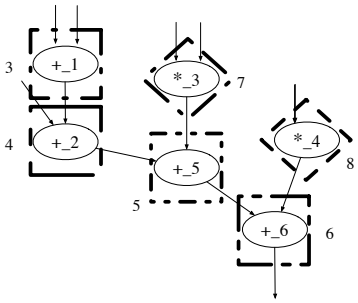


Figure 4: Matches for instruction 2 and 3, shown in dashes.

To find all possible matches of the instructions on the application graph, we record all different matches of the instruction graphs on the application graph. A match is identified if it is sub-graph isomorphic to the application graph. In our case, a graph h is sub-graph isomorphic to a target graph g if each node and edge in h matches a node and edge in a sub-graph of g. We introduce this as a set of constraint satisfaction problems (CSP) and find all matches for each instruction. For each instruction h, we record its matches in the set $matches_h$.

Input: Application graph g, set of instruction graphs I
  for all $h \in I$ **do**
    $matches_h = subGraph\_CSP(g,h)$
  end for

In the above, $subGraph\_CSP(g,h)$ refers to the method that generates a CSP for the sub-graph isomorphism problem for application graph g and instruction graph h, and returns *all possible solutions* to it. Each node in the instruction graph gets a finite domain variable (FDV) that is assigned to the id of the node in the application graph it matches to, i.e. when an isomorphic sub-graph is found in g for h (see section 2).

4.3 Instruction Selection and Scheduling

After finding all possible matches, we can continue with selecting the matches that will actually be used and schedule them to get the shortest schedule. Match selection and scheduling are not independent from each other. Some match selections may lead to shorter schedules, while shorter schedules may enforce a particular match selection. Therefore, in a CSP context, solving these two dependent prob-

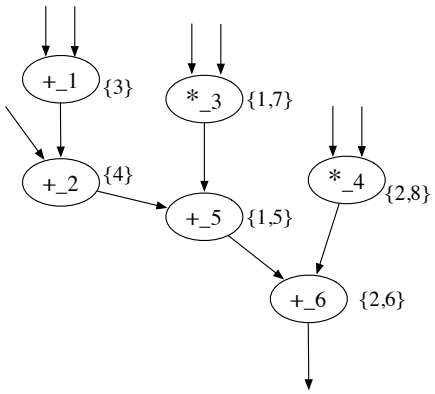


Figure 5: Initial domains for the *nodeMatch* variables for each node is shown in brackets.

lems simultaneously is likely to result in a better pruning in both match selection and scheduling domains, hence a smaller search space.

Target platforms with different properties (e.g. number of cores, instructions available) require a tailored constraint model, while there is a set of constraints that are generic for instruction selection and scheduling problem, regardless of the target platform. We first describe the generics, then continue with problem specific ones. In the following mathematical notations, i denotes the instruction index and m denotes the match index for the respective instruction, and they quantify over all instructions and their matches.

4.3.1 Generic Constraints

While selecting instruction matches, no node in the application graph should be left uncovered. To keep track of which node is covered by which match, we use a vector of FDVs named *nodeMatch*, where $nodeMatch_n$ refers to the match number that node with id n from the application graph is covered by. The initial state of *nodeMatch* for our running example is depicted in Fig. 5. Note that the domain of each variable is the same as the domain of match ids. Therefore any solution to the CSP with the *nodeMatch* variables assigned to one value, means that the graph is fully covered.

As our problem definition includes scheduling, we use a FDV for each node which denotes the start time of that node, i.e. when it is scheduled to run. The start times of the nodes within a selected instruction match should be equal since they are to be run as one instruction. In constraint (1) (and the following), *matches* denotes a three dimensional vector that keeps the relation between the nodes of the matches of the instructions, and the nodes of the application graph. As mentioned

before, an instruction graph can have multiple matches on different subgraphs of the application graph. With this information in mind, $matches_{i,m,p}$ points to the id of the application graph node, that node p of match m of instruction i matches to. To illustrate this, $matches_1 = [[4,6],[3,5]]$ for our example, which corresponds to the matches of the first instruction (see Fig. 3).

We introduce also the matrix named *sel* that keeps boolean domain variables for each instruction match, denoting whether or not they are selected to cover the application graph (see constraints (5) and (6) for more details). Together with *matches*, *sel* lets us reason about entire matches rather than particular nodes. Using these two vectors, constraint (1) states that if two nodes can be covered by the same instruction match and if the respective match is selected, then their start times will be equal.

$$\begin{aligned} &\forall p,q \in nodes \mid p \neq q; p,q \in matches_{i,m} : \\ &\quad sel_{i,m} \Rightarrow start_p = start_q \end{aligned} \tag{1}$$

Constraint (2) sets the durations of nodes that have output edges to other instructions (namely, the output nodes) to $\triangle t$ (the duration of the respective instruction), forcing the destination of the output edge to wait until this instruction is finished (together with constraints (3, 4)). Duration of a non-output node is set to 0 since it actually does not cause any wait when scheduling. Constraints (3, 4) on the other hand, impose the precedence constraints caused by any edge (p,q) from node p to node q in the application graph. Nodes that are not in the same match are handled by constraint (4) while constraint (3) handles the nodes in the same match. Note that constraint (3) ignores precedence between the nodes in a *selected* match.

$$\begin{aligned} &\forall p \in nodes \mid p \in matches_{i,m} : \\ &(sel_{i,m} \wedge p \in outputs_{i,m}) \Rightarrow duration_p = \triangle t \\ &\qquad \wedge \\ &(sel_{i,m} \wedge p \notin outputs_{i,m}) \Rightarrow duration_p = 0 \end{aligned} \tag{2}$$

$$\begin{aligned} &\forall (p,q) \in edges \mid p,q \in matches_{i,m} : \\ &\neg sel_{i,m} \Rightarrow start_p + duration_p \leq start_q \end{aligned} \tag{3}$$

$$\begin{aligned} &\forall (p,q) \in edges \mid p,q \notin matches_{i,m} : \\ &\quad start_p + duration_p \leq start_q \end{aligned} \tag{4}$$

To understand constraint (3) better, consider Fig. 6, where $instruction_1$ has a multiplication with two outgoing edges: one to the subtraction in the same instruction, another one as an output of the instruction. The only possible matching for

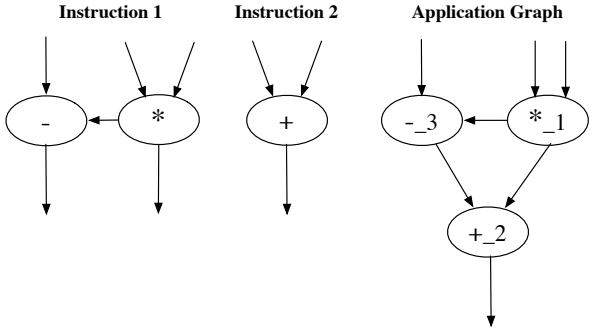
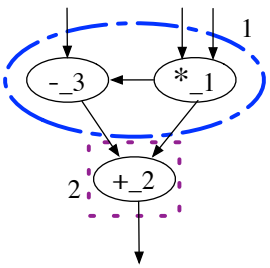


Figure 6: Example with an instruction that has an output node that has an outgoing edge to another node in the instruction.

this example is depicted in Fig. 7, therefore to reach a full cover, these matches have to be selected (i.e. $sel_{1,1} = sel_{2,1} = 1$). If each instruction has duration $\triangle t$, the shortest schedule on a processing unit that runs one instruction at a time will be $2 \triangle t$. By constraint (2), having an outgoing edge outside the instruction, both $duration_1$ and $duration_3$ will be $\triangle t$. If the precedence between nodes in a selected match is not neglected as in constraint (3), $node_3$ will have to wait for $node_1$ to finish, adding an additional $\triangle t$ duration to the whole $match_1$. Since $node_2$ is dependent on the output of $node_3$, this extra duration will result in a schedule with length $3 \triangle t$ instead of $2 \triangle t$. By means of constraint (3), we ignore the dependency between $node_1$ and $node_3$ and eliminate this problem.

nodeMatch and *sel* variables are logically bound to each other since $nodeMatch_n$ denotes which match covers node n in the application graph, while $sel_{i,m}$ denotes if the match indexed as i,m is selected or not. So, logically, if a match k is selected, *all* the nodes that can be covered by this match should have their *nodeMatch* variable set to k (denoted in constraint (5)). Similarly, if k is not selected, *none* of the nodes that can be covered by this match can have k as their *nodeMatch* variable (denoted in constraint (6)). In mathematical notation, constraint (5) and constraint (6) describes this logic. Note that *nodeMatch* keeps match numbers for practical reasons, therefore we need a flattened intermediate representation of the *matches* matrix, which is denoted as $\forall k \mid match_k = matches_{i,m}$. Both constraints are implemented using the global




```
    node1 precedes node3 ⇒ |schedule| = 3△t
¬ (node1 precedes node3) ⇒ |schedule| = 2△t
```

Figure 7: Only possible matching for the example in Fig. 6.

Count constraint and *reification* to reach an effective implementation.

$$\forall k \mid \; match_k = matches_{i,m} : \bigwedge_{p \in match_k} nodeMatch_p = k \iff sel_{i,m} \tag{5}$$

$$\wedge$$

$$\bigwedge_{p \in match_k} nodeMatch_p \neq k \iff \neg sel_{i,m} \tag{6}$$

A valid schedule does not exceed the resource limit at any time. In our case, the resource limit is the number of processing units available in the given architecture. For this purpose CP offers a well-studied global constraint named `Cumulative`, that is used commonly for task scheduling problems [8, 19] (see section 2).

Matches in our problem are task candidates, i.e. if they are selected, they are to be scheduled with respect to previously mentioned constraints, plus the cumulative constraint. The start times of the nodes in the same selected match are bound together by constraint (1). Thus, scheduling one representative node for each selected match will also mean scheduling the whole application, given that the application is covered by the selected matches, (5) and (6). Therefore, we pick the first node (any node would do) in every $match_m$ as the representative and add its start time to the list of task starts (*mStarts* in constraint (7)) for the cumulative constraint. To consider only the selected matches in scheduling, we multiply $sel_{i,m}$ with the duration of the $instruction_i$ (that is $\triangle t$ in this paper) that $match_m$ belongs to, and save this as the duration for $match_m$ (*mDurations* in constraint (7)). Non-selected matches, which will eventually get zero duration, will have no effect on the schedule. Since every instruction is run on only one processing unit, and processing units run one instruction at a time, resource need for each match is 1 (a

vector of *ones* in constraint (7)). Finally, resource limit that is not to be exceeded is number of processing units available (*nProcs* (7)).

$$\texttt{Cumulative}(mStarts, mDurations, ones, nProcs) \tag{7}$$

4.3.2 Architecture/Problem Specific Constraints

Different architectures have different properties. In our problem definition, they may have different number of processing units running in parallel; and the set of possible instructions and the interaction between those instructions can be different. The number of processing units is simply parameterized. Properties involving instruction sets on the other hand, pose a harder problem that requires more attention.

Sleipnir architecture, for example, has two instructions, named $idct8pfw$ and $idct8pbw$, that combined together, implement IDCT with 8 inputs. As seen in Fig. 1 and 11, both instructions have two disjoint connected components. Matching such instructions on big application graphs, e.g. comprising 8 parallel 8-input IDCTs (see section 5), will result in a combinatorial explosion of matches. This is caused by disjoint connected components in the same instruction.

An illustrative example is depicted in Fig. 8. The only instruction in the example includes two disjoint connected components (both components have only one node for simplicity). The application on the other hand, is a parallel repetition of the instruction. We use the labels to count the instances of the instruction where n represents the total number of instances. For $n = 3$, the matches for $instruction_1$ would look like in Fig. 9. All the matches from $match_4$ to $match_9$ can be considered as side effects of the fact that $instruction1$ has disjoint connected components. If the instruction included an edge connecting the disjoint components as in Fig. 10, the total number of matches would be 3 instead of 9 (providing that the connecting edge is replicated in the application graph too).

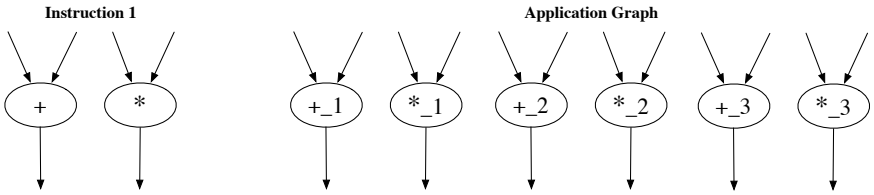


Figure 8: A problem instance with an instruction with disjoint connected graphs and its parallel instances as the application

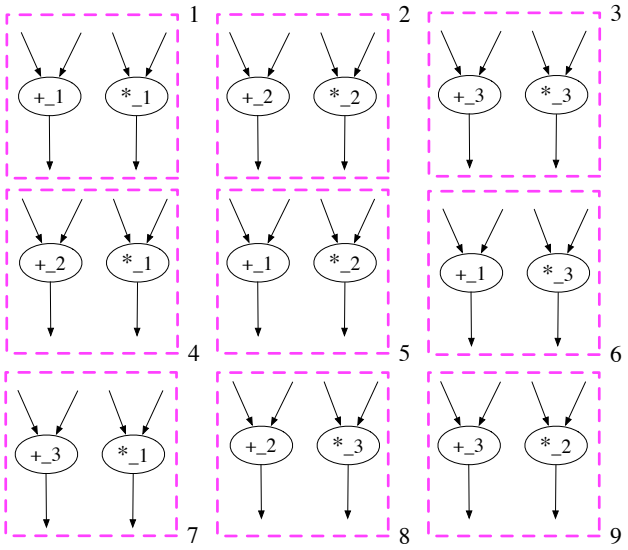


Figure 9: Instruction matches for the example in Fig. 8.

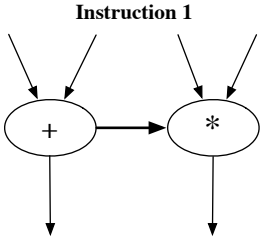


Figure 10: Ideal instruction for the example problem in Fig. 8

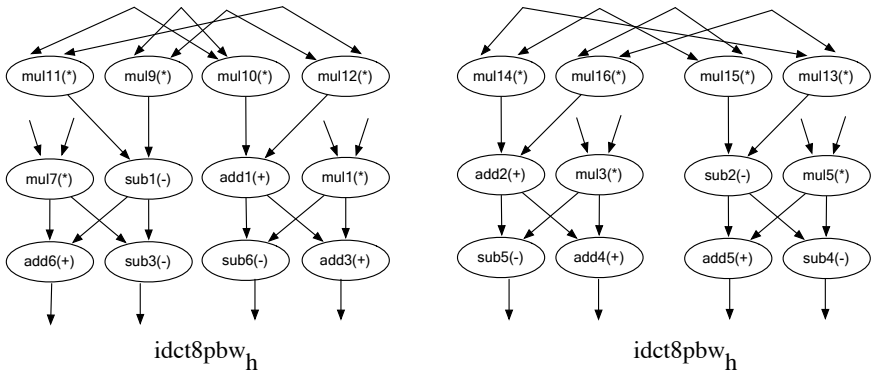


Figure 11: *idct8pbw*: Sleipnir instruction that covers the second half of Loeffler's IDCT algorithm [18]

If we generalize the problem over an instruction with p disjoint connected components, and the application as n parallel replications of this instruction, the number of matches becomes n^p, instead of n, which happens when the instruction consists of one connected graph.

To cope with this combinatorial explosion, we divide the disjoint components in the instruction graphs and represent them as individual instructions. This way, the total number of matches are reduced from n^p to np, where $p = 2$ for the IDCT instructions in Sleipnir. However, we still have to ensure that the instructions that we divided into two are actually merged in the schedule, i.e. instruction i is divided into i_a and i_b, each i_a has to be run at the same time with a corresponding i_b. Another side effect is that, the number of processing units that are modeled has to be doubled, in order to represent the half of the Sleipnir core that runs i_a and the other half that runs i_b. Otherwise these half instructions will be sequentialized, which breaks the atomicity of instructions.

The second instruction for implementing IDCT named *idct8pbw*, depicted on fig. 11, comprises two disjoint connected components that are actually isomorphic with each other. Therefore, representing this instruction as two isomorphic instructions will simply double their matches. Instead, we let one half instruction to represent both of them. Again, we have to ensure that the matches of this half instruction are merged into a full instruction correctly.

As a result of dividing the instructions $idct8pfw$ and $idct8pbw$ as mentioned above, we get two different instructions that together represent $idct8pfw$: $idct8pfw_a$ and $idct8pfw_b$; and $idct8pbw_h$ that represents half of $idct8pbw$. As for the division of the processing core, each Sleipnir core will have two parts: one that can only run either $idct8pfw_a$ or $idct8pbw_h$, another one that can only run either $idct8pfw_b$ or $idct8pbw_h$.

With all this information, possible match groupings that can run on the same processing unit simultaneously are: $\{idct8pfw_a, idct8pfw_b\}$,$\{idct8pbw_h, idct8pbw_h\}$

The second grouping above means that two matches of $idct8pbw_h$ can run simultaneously at the same core, since two halves comprise a complete $idct8pbw$. For these groupings, we generate start time and duration lists similar to the lists for constraint (7), as illustrated in the algorithm below, for the $\{idct8pfw_a, idct8pfw_b\}$ grouping. As in constraint (7) we pick the first node in a match as its representative. This is why we add $start_{matches_{i,m,0}}$ to the respective $mStarts$ list. Note that $\oplus$ is used as list concatenation operation and $matches_{idct8pfw_a}$ represents all the matches for the $idct8pfw_a$ instruction.

Input: $mStarts_{ab} = \emptyset$, $mDurations_{ab} = \emptyset$
  if $matches_{i,m} \in matches_{idct8pfw_a} \cup matches_{idct8pfw_b}$ **then**
    $mStarts_{ab} \Leftarrow mStarts_{ab} \oplus start_{matches_{i,m,0}}$
    $mDurations_{ab} \Leftarrow mDurations_{ab} \oplus sel_{i,m} * \triangle t$
  end if

On the other hand, only one match of $idct8pfw_a$ can be run at one core, until it is finished. Same rule applies to $idct8pfw_b$. Therefore, we generate similar lists for $matches_{idct8pfw_a}$ and $matches_{idct8pfw_b}$ i.e. $mStarts_a$, $mDurations_a$ and $mStarts_b$, $mDurations_b$ respectively. After generating these lists, the combination of the constraints (8), (9), (10) and (11) ensures that the grouped matches can run simultaneously on the same core by multiplying the resource limit by 2 ((8, 9)); while no more than one match of the same instruction for $idct8pfw_a$ or $idct8pfw_b$ can be run on the same core ((10, 11)).

$$\texttt{Cumulative}(mStarts_{ab}, mDurations_{ab}, ones, nProcs * 2) \tag{8}$$

$$\texttt{Cumulative}(mStarts_h, mDurations_h, ones, nProcs * 2) \tag{9}$$

$$\texttt{Cumulative}(mStarts_a, mDurations_a, ones, nProcs) \tag{10}$$

$$\texttt{Cumulative}(mStarts_b, mDurations_b, ones, nProcs) \tag{11}$$

As mentioned earlier, dividing the original instructions into two parts required a modeling style that doubles the number of available processing units, in order to simulate a Sleipnir core's capability to run the whole original instruction at once. Thus, two processing units (after doubling) actually represent one core. Based on this assumption, we change the $nProcs$ in constraint (7) to $nProcs * k$, where k is the number of divisions of the instructions (in Sleipnir case $k = 2$) as shown in constraint (12).

$$\texttt{Cumulative}(mStarts, mDurations, ones, nProcs * k) \tag{12}$$

We do not explicitly define a constraint for assuring that each half instruction match has a corresponding second half, since the combination of the constraints (8), (9), (10) and (11) together with the minimization of the schedule length (see section 4.5) results in a schedule where each match grouping mentioned above are run simultaneously. Note that this is only valid when the original non-divided instructions (i.e. $idct8pfw$ and $idct8pbw$) can cover the application graph. In other cases additional constraints are defined.

The fact that $idct8pbw$ is divided into two isomorphic graphs creates problems that are not addressed by the constraints mentioned above. As they are parts of originally different instructions, $idct8pbw_h$ and any one half of $idct8pfw$ ($idct8pfw_a$ or $idct8pfw_b$) can not be merged together i.e. run on the same core simultaneously. A limiting constraint such as constraint (13) can not be used in this case, since it would render constraint (9) useless. This is based on the fact that $mStarts_{ah}$ and $mDurations_{ah}$ would include $mStarts_h$ and $mDurations_h$ already, and limit them with $nProcs$ instead of $nProcs*2$, which eventually prohibits them to run on the same core simultaneously. To solve this, we need to involve resource assignment constraints instead of using only scheduling constraints that disregard specific resource assignment, such as `Cumulative`. We describe this further in section 4.4.

$$\texttt{Cumulative}(mStarts_{ah}, mDurations_{ah}, ones, nProcs) \tag{13}$$

So far, we only considered the Sleipnir architecture for architecture/problem specific constraints. A similar modeling style is used to solve the problem for VLIW. Although this time, we focus on the groups of matches that can not be run simultaneously on the same core i.e. only one of the matches in the group can be run on a core:{*add*, *sub*},{*mul*}, {*ld*, *str*}. Again, we generate respective lists for start times and durations as in the previous algorithm and limit the maximum simultaneous execution to *nProcs*. So, for each group in the list above, we have one corresponding constraint (10) with respective lists for start times and durations generated with the given algorithm. On the other hand we change the constraint (7) to constraint (12) with $k = 3$ since an instruction is divided into three in our VLIW model (see section 4.1).

4.4 Resource Assignment

In the beginning of the previous section, we tied the start times of the nodes that are covered by the same selected match using constraint (1). Besides being logical, this let us use one representative node for each match and schedule matches instead of separate nodes. To use the same approach, we define the basic resource constraints in constraint (14), where each node in a selected match is assigned to the same resource.

$$\forall p,q \in nodes \,|\, p \neq q; p,q \in matches_{i,m} : \\ sel_{i,m} \Rightarrow resource_p = resource_q \tag{14}$$

Series of `Cumulative` constraints guarantee only the existence of a valid schedule given the number of processing units available, but without giving any information regarding the resource assignment. The problem arises from the fact that the scheduling constraints, e.g. `Cumulative` constraints, do not constrain the resource assignment variables *for each match*, which we denote as *mResources*. These variables represent the resource that the selected match is assigned to.

To eliminate any invalid resource bindings, we use the `Diff2` global constraint which is described in section 2 as seen in constraint (15). Note that $mDurations_m$ becomes zero when the match is not selected, as discussed when describing constraint (7).

$$\texttt{Diff2}(mStarts, mResources, mDurations, ones) \tag{15}$$

Individual matches are assigned to the processing unit that is specific for the instruction of the respective match, whenever possible. For example, considering VLIW, we model each core as divided into three processing units (*pu*), where pu_1 runs *add*/*sub* matches, pu_2 runs *mul* matches and finally pu_3 runs *ld*/*str* matches. Resource variable for an *add* match on the other hand can get a value from $\{1, k+1, 2k+1, ..., k(P-1)+1\}$ where k is the number of divisions of a core and P is the total number of cores available. The constraint embodying this idea is given below. Similar constraints for the other two instructions are used for the VLIW architecture.

$$m \in matches_{add/sub} : \\ resource_m \bmod k = 1 \tag{16}$$

In case of Sleipnir, matches of $idct8pfw_a$ are allocated to the first halves of the cores ($resource_a \in \{1,3,5,...,2(P-1)+1\}$) and $idct8pfw_b$ ($resource_b \in \{2,4,...,2(P)\}$) matches to the second. However a valid schedule with a valid resource assignment is still not guaranteed because the halves of $idct8pbw$ are isomorphic, as discussed in the previous section. It is not possible to allocate these halves to a particular half of the core either, since they can actually be run on both. Instead, for each match of $idct8pfw_a$ and $idct8pfw_b$, we prohibit having a match of $idct8pbw_h$ that is selected and starts at the time at the same core (see constraint constraint (17)). Note that since $matches_{idct8pfw_a}$ are assigned to the first half of the core and $matches_{idct8pfw_b}$ are assigned to the second half; we prohibit

$resource_a = resource_h - 1$ for $matches_{idct8pfw_a}$ and $resource_b = resource_h + 1$ for $matches_{idct8pfw_b}$. Assignment to the same processing unit at the same time is already prohibited by constraint (15).

$$\nexists h \in matches_h : a \in matches_{idct8pfw_a} \wedge$$
$$sel_h = sel_a \wedge start_h = start_a \wedge resource_h - 1 = resource_a \quad (17)$$

4.5 Search Space Heuristics

Our problem definition includes finding the shortest schedule (or one such schedule in case several shortest schedules exist). The completion of a node is defined as $start_n + duration_n$ and minimizing the maximum of this completion for all sink nodes (i.e. nodes with no successors) gives the shortest schedule.

As most consistency techniques are not complete, 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 or heuristic 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.

Even though the constraint model is unified, we divide the search into three sequential phases (see Fig. 12): Instruction selection, scheduling and resource assignment 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. Each phase has a set of variables to pick from, represented as a vector (e.g. $nodeMatch_{matches}$ in the instruction selection phase). Note that these vectors include one representative variable for each match. Since each node is already tied to the other nodes in the same selected match through constraints (1) and (14), one variable per match is enough.

The first two phases find the shortest schedule without resource assignment (i.e. an ordering of selected matches), using a branch-and-bound search with backtracking. If such a schedule exists and is found, according to the constraints defined in section 5, a valid resource assignment has to exist. The third phase of the search finds one such assignment and ends the search procedure.

The decision of the which match to cover a node directly affects its start time (*start* variable) and the resources it can be run on (*resource* variable). Therefore, the first phase searches for a valid instruction selection and involves the vector

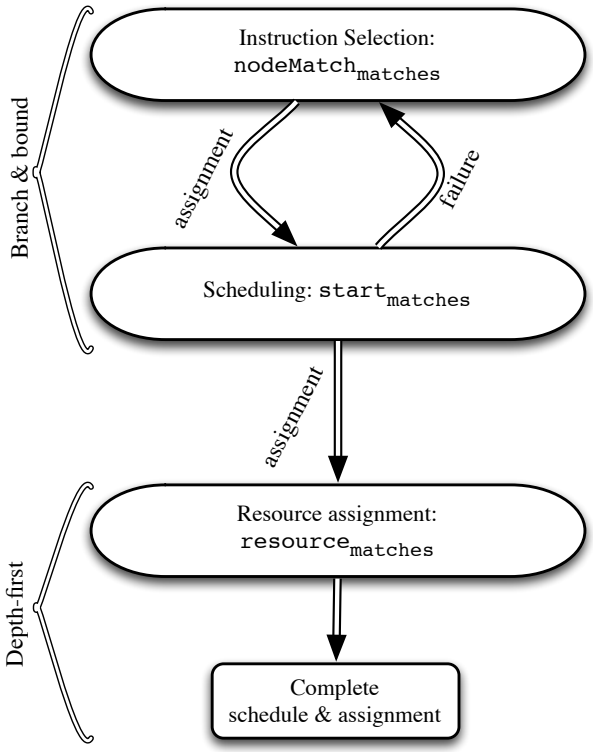


Figure 12: Sequential search

$nodeMatch_{matches}$, that comprises one representative *nodeMatch* variable for each match.

When a valid assignment for the $nodeMatch_{matches}$ vector is found, the search moves on to the second phase where the start times for the matches is the decision point. A failure in this level means a valid schedule for the selection does not exist, and the solver backtracks to the instruction selection level. Whenever we need to pick the next variable from $start_{matches}$ and assign a value to it, we pick the one with the smallest maximum value and set it to the minimum value in its domain. The idea behind this combined heuristic is to decide on the start times in the order they are to be scheduled and assign the minimum possible value first to find the shortest schedule faster.

As mentioned above, when the first two phases are finished, the only thing left is a valid resource assignment, which is guaranteed to exist by respective cumulative and resource assignment constraints. Therefore this is left as the last phase.

Again, we use a vector that includes one resource assignment variable per match, shown in Fig. 12 as $resource_{matches}$.

5 Experiments

In our experiments, we considered several application graphs on different architectures. Our main interest in the Sleipnir architecture [1] is the special instructions devised for IDCT and DCT. Therefore, we used an application graph for an implementation of Loeffler's IDCT [18]. To experiment with different problem sizes and number of Sleipnir cores available, we consider IDCTs sizing up to 8x8, i.e. eight parallel copies of single IDCT, and up to four Sleipnir cores (only for 8x8 IDCT).

For VLIW and RISC architectures, we conducted a series of experiments with some of the benchmark graphs from ExpressDFG [14] which provides dataflow graphs for benchmark applications from MediaBench [20] (among others). Single IDCT used for Sleipnir experiments is also included. To evaluate the scalability of our method, we have also scheduled these applications on several cores for both architectures.

Table 1 shows the results for experiments on VLIW architecture. The name of the benchmark and the characteristics of the application graph is summarized in the first column, including number of nodes (|V|), number of edges (|E|) and length of the critical path. Together with these, we show the number of FDV's and constraints generated for the CSP model. We experiment also with the available number of cores. Not all of the experiments for VLIW resulted in a solution that is proven optimal. This means that a solution is found, but the solver timed-out before finishing the search for a better solution. The time-out is set at 5 minutes.

For the cases that reached the time-out, we recorded the last solution, and ran the solver again with this solution as the lower bound to get the "runtime" (see the column "runtime" for the rows with "no" for the "proved optimal" column). Otherwise, "runtime" represents the runtime of the solver.

We also include a lower bound estimation for problem instances that are not proved optimal, in order to assess how close we are to the optimal solution. The lower bounds are estimated in the following fashion. First we calculate the theoretical lower bound i.e. the maximum of critical path length of the graph and the number of clock cycles it would take to run the application if the dependencies between operations were ignored. Once we have this theoretical lower bound we try to improve it through experimentation. We run our solver with setting the schedule length to the lower bound with a timeout of 15 minutes. If the solver figures out a contradiction within the time limit, it means that there is no solution with this schedule length, thus we increase the lower bound with one. This operation is repeated until the solver returns a solution or times out. A solution here means the optimal solution while a time out means that the solver could not find a solution or

Benchmark {|V|, |E|, Cr. Path} {# FDV's, # const}	# cores	schedule length	proved optimal	lower bound	runtime (ms)
IDCT	1	54	no	53	2141
{96, 160, 4}	2	27	yes	-	1902
{495, 527}	4	17	no	15	1059
MESA Matrix Mul.	1	45	yes	-	2153
{109, 116, 9}	2	23	yes	-	1872
{448, 455}	4	12	yes	-	1242
Elliptic Wave Filter	1	27	yes	-	540
{34, 47, 14}	2	16	yes	-	416
{148, 161}	4	14	yes	-	327
Cosine1	1	31	no	28	636
{66, 76, 8}	2	17	no	16	514
{279, 289}	4	10	yes	-	34153
MESA Smooth Tri.	1	86	no	81	17102
{197, 196, 11}	2	44	no	41	8338
{804, 803}	4	23	no	22	5120

Table 1: Benchmark applications on VLIW

a contradiction in given time. The schedule length at the final iteration is recorded as the lower bound.

According to this estimation, in the cases where the time-out is reached, the solutions are on average 6 % off the lower bound, where in the worst case the rate is 13 % (i.e. IDCT with 4 cores).

Table 2 shows the results for RISC architecture. For the Sleipnir architecture with IDCT we present the results of the experiments in Table 3, with graphs of different sizes and with different numbers of available cores (only for 8x8 IDCT). For both tables all the results were proven optimal.

To see how our model performs with structurally different graphs, we picked benchmarks with varying number of nodes, length of critical paths, and other characteristics such as number of connected components. These variances also help identify the characteristics that hinder the solver performance, as in VLIW case.

Benchmark {|V|, |E|, Cr. Path} {# FDV's, # const}	# cores	schedule length	runtime (ms)
IDCT	1	96	833
{96, 160, 4}	4	24	1695
{701, 637}	16	14	856
MESA Matrix Mul.	1	109	782
{109, 116, 9}	4	28	917
{659, 557}	16	9	1009
Elliptic Wave Filter	1	34	392
{34, 47, 14}	4	14	333
{209, 188}	16	14	315
Cosine1	1	66	639
{66, 76, 8}	4	17	628
{401, 345}	16	8	656
MESA Smooth Tri.	1	197	1903
{197, 196, 11}	4	50	1859
{1187, 989}	16	14	2207

Table 2: Benchmark applications on RISC

IDCT size {|V|, |E|, Cr. P}	# cores	schedule length	runtime (ms)	# FDV's	# const
1x8 {96, 160, 4}	1	4	484	681	682
2x8 {192, 320 4}	1	8	909	1413	1478
3x8 {288, 480, 4}	1	12	941	2209	2402
4x8 {384, 640, 4}	1	16	1038	3069	3354
8x8 {768, 1280, 4}	1	32	1691	7149	8942
	2	16	1976	7149	8942
	4	8	1845	7149	8942
	8	4	1742	7149	8942

Table 3: Sleipnir experiments with IDCT

Benchmark {|V|, |E|, Cr. Path} {# FDV's, # const}	# cores	schedule length	runtime (ms)
Elliptic Wave Filter	1	23	631
{34, 47, 14}	2	13	513
{149, 173}	4	12	425

Table 4: Experiments on VLIW, extended with a custom instruction

Our model is intended to handle VLIW-like architectures, where part of the instruction is a complex instruction itself. To demonstrate such a case, we included the instruction depicted in Fig. 13 in the VLIW. We let this complex instruction to be run in the load/store slot of the VLIW and assume that it takes one clock cycle to execute. We experimented only on the elliptic wave filter benchmark with various number of available cores. The results are presented in Table 4. All the results in this table are proved optimal. Cases where the schedule length is shorter than the critical path is a side effect of the fact that the complex instruction is part of the critical path.

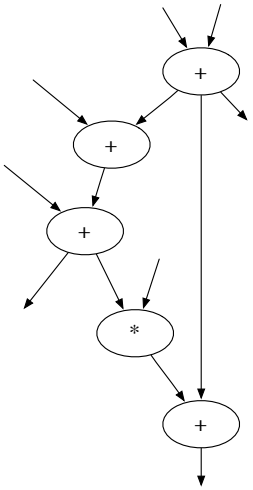


Figure 13: Complex instruction to extend the VLIW model

6 Discussion and Future Work

Previous section showed that our model solves the RISC and Sleipnir problems optimally while VLIW architecture poses a bigger problem. Only almost half of the problem instances are solved optimally. While optimality generally is not a requirement in mapping DSP applications, it identifies some problem instances harder than others in our case. The applications named "Cosine1" and "MESA Smooth Triangle" resulted in solutions that are not proven optimal. The main reason for this is the fact that both application graphs consist of disjoint connected components. Since there is no precedence required between these components, any ordering of them or their sub-components results in a valid unique schedule with the same schedule length. This means that while searching for the shortest

schedule, the solver has to go through all re-orderings of these *mutually independent* components, many times.

IDCT graph as a whole is a connected component, but parts of it are still mutually independent of each other (see Fig. 1 and 11). Therefore, the solver faces the same problem described above for "Cosine1" and "MESA Smooth Triangle". On the other hand, the solver's ability to prove optimality for 2 cores but not 4 cores is peculiar and subject to further investigation.

Heffernan and Wilken [21] proposed a method for alleviating this problem of mutually independent components. In particular they introduce *latency-zero edges* between nodes that are mutually independent. Such edges are added only when they do not change the optimality. This way, the re-ordering of the nodes connected with the new edge is disabled. This technique is not directly applicable to our method since our graph representation does not include latency on edges. Furthermore, we schedule instruction matches and an instruction can consist of several nodes. This causes problems in finding suitable pairs of nodes for connecting with a latency-zero edge. We plan to address these issues and adapt this technique to our method.

Searching for a solution in an NP-hard problem space requires sophisticated search methods. Our search strategy uses standard heuristics and techniques provided by the constraint solver. These techniques perform well in general cases and can be customized to some extent to fit the problem better. Even so, they are too generic for the VLIW case as our results hint. Therefore, we intend to devise a custom search method which incorporates (among others) analysis of the application graph such as *critical path* and *slack* analysis.

Our current work focused on instruction selection and scheduling, disregarding memory and register allocation constraints. Together with the improvements mentioned above, this is the main problem we plan to tackle in the near future. This will let us generate code and possibly integrate our method in a larger system.

7 Conclusions

We presented a constraint based approach for instruction selection and scheduling problem. Our results show that it is a technique worth investigating further. We solve most problem instances optimally, together with some exceptional cases where we still find a valid solution within a short period of time. The flexibility of constraint programming enables plugging architecture or problem specific constraints in and out of the model very easily. This makes it possible to use the same core model for different architectures.

We experimented on three architectures that have different capabilities and instruction structure. As the simplest case, we used a RISC architecture with varying number of cores. We also worked on a custom architecture, namely Sleipnir, where instructions are tailored for DSP applications and can comprise more than 20 ba-

sic operations i.e. addition, subtraction, multiplication. Finally, we experimented with a VLIW architecture where each partition of the VLIW can be a complex instruction itself.

Identifying characteristics for application and instruction graphs that hinder proving optimality of a solution is an open question we plan to investigate further. Accompanied with memory and register allocation constraints, our model can generate performance efficient code for DSP kernels on custom architectures.

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

References

[1] Dake Liu, Andreas Karlsson, Joar Sohl, Jian Wang, Magnus Pettersson, and Wenbiao Zhou. "ePUMA Embedded Parallel DSP Processor with Unique Memory Access". In: *ICICS, 8th International Conference on Information, Communications and Signal Processing*. 2011.

[2] Johan Eker and Jörn W. Janneck. *CAL language reference*. Tech. rep. UCB/ERL M03/48. Electronics Research Lab , University of California, Berkeley, 2003.

[3] Apostolos A. Kountouris and Christophe Wolinski. "Efficient scheduling of conditional behaviors for high-level synthesis". In: *ACM Trans. Des. Autom. Electron. Syst.* 7.3 (2002), pp. 380–412.

[4] Ruirui Gu, Jörn W. Janneck, Mickaël Raulet, and Shuvra S. Bhattacharyya. "Exploiting statically schedulable regions in dataflow programs". In: *ICASSP 2009. IEEE International Conference on Acoustics, Speech and Signal Processing*. April 2009, pp. 565–568.

[5] Mehmet Ali Arslan and Krzysztof Kuchcinski. "Instruction Selection and Scheduling for DSP Kernels on Custom Architectures." In: *DSD*. IEEE, 2013, pp. 821–828.

[6] Krzysztof Kuchcinski. "Constraints-Driven Scheduling and Resource Assignment". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 8.3 (July 2003), pp. 355–383.

[7] Willem-Jan van Hoeve and Irit Katriel. "Handbook of Constraint Programming". In: Foundations of Artificial Intelligence. Elsevier Science, 2006. Chap. Global Constraints.

[8] Abderrahmane Aggoun and Nicolas Beldiceanu. "Extending chip in order to solve complex scheduling and placement problems". In: *Mathematical and Computer Modelling* 17.7 (1993), pp. 57–73.

[9] Nicolas Beldiceanu and Evelyne Contejean. "Introducing global constraints in CHIP". In: *Journal of Mathematical and Computer Modelling* 20.12 (1994), pp. 97–123.

[10] Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch, and François Charot. "Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation". In: *ACM Trans. on Reconfigurable Technology and Systems (TRETS)* 5.2 (June 2012), 10:1–10:38.

[11] Chris Lattner and Vikram 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.

[12] Andrzej Bednarski and Christoph Kessler. "Integer Linear Programming versus Dynamic Programming for Optimal Integrated VLIW Code Generation". In: *12th Int. Workshop on Compilers for Parallel Computers*. 2006.

[13] Peter van Beek and Kent Wilken. "Fast optimal instruction scheduling for single-issue processors with arbitrary latencies". In: *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*. Paphos, Cyprus, November, 2001, pp. 625–639.

[14] ExpressDFG. *ExpressDFG benchmark website.* 2006. URL: `http://express.ece.ucsb.edu/benchmark`.

[15] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. "Constraint-based Register Allocation and Instruction Scheduling". In: *Eighteenth International Conference on Principles and Practice of Constraint Programming*. Québec City, Canad, Oct. 8-12, 2012.

[16] Erwan Raffin, Christophe Wolinski, François Charot, Krzysztof Kuchcinski, Stéphane Guyetant, Stéphane Chevobbe, and Emmanuel Casseau. "Scheduling, binding and routing system for a run-time reconfigurable operator based multimedia architecture". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. Edinburgh, UK, Oct. 26-28, 2010.

[17] Abid M. Malik, Jim McInnes, and Peter van Beek. *Optimal basic block instruction scheduling for multiple-issue processors using constraint programming*. Tech. rep. In: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence, 2005.

[18] Christoph Loeffler, Adriaan Ligtenberg, and George S. Moschytz. "Practical fast 1-D DCT algorithms with 11 multiplications". In: *ICASSP-89, International Conference on Acoustics, Speech, and Signal Processing,* May 1989, 988–991 vol.2.

[19] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International series in operations research & management science. Kluwer Academic, 2001. URL: `http://books.google.se/books?id=mhK-V1VeWYgC`.

[20] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems". In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Research Triangle Park, North Carolina, USA, 1997, pp. 330–335.

[21] Mark Heffernan and Kent Wilken. "Data-dependency graph transformations for instruction scheduling". In: *Journal of Scheduling* 8 (2006), p. 2005.

PAPER II

PROGRAMMING SUPPORT FOR RECONFIGURABLE CUSTOM VECTOR ARCHITECTURES

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.

Mehmet Ali Arslan, Krzysztof Kuchcinski and Flavius Gruian - Dept. of Computer Science, Lund University
Yangxurui Liu - Electrical and Information Tech., Lund University

*Based on the paper published in the proceedings for PMAM 2015: The 6th International Workshop on Programming Models and Applications for Multicores and Manycores, February 07 - 11, 2015, San Francisco, CA, USA, 2015

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.

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 run-time 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.

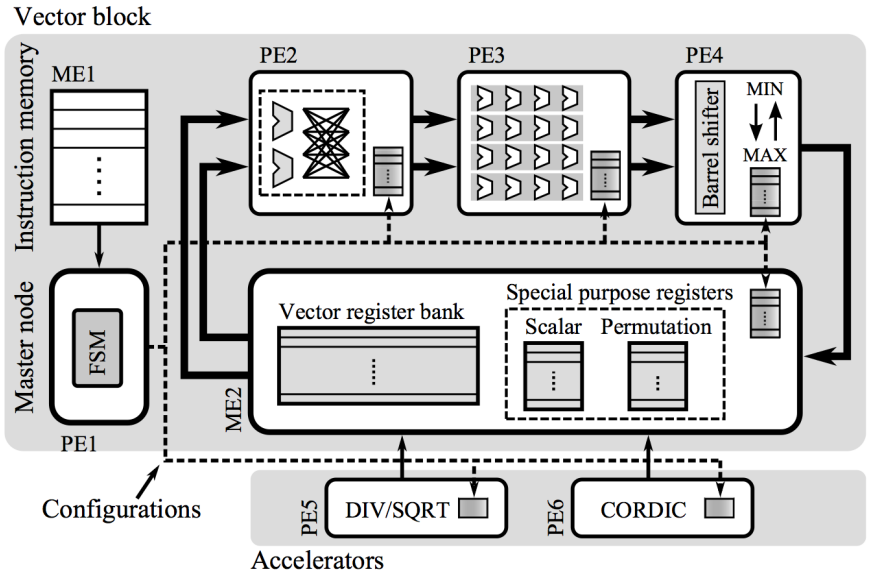


Figure 1: Micro-architecture of the vector processor, consisting of 6 PEs and 2 MEs. Solid and dashed lines depict data and control bus, respectively.

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 Constraint Programming

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.

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 multi-processor 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 sub-graph isomorphism. In this work we focus less on instruction selection and more on scheduling with memory allocation.

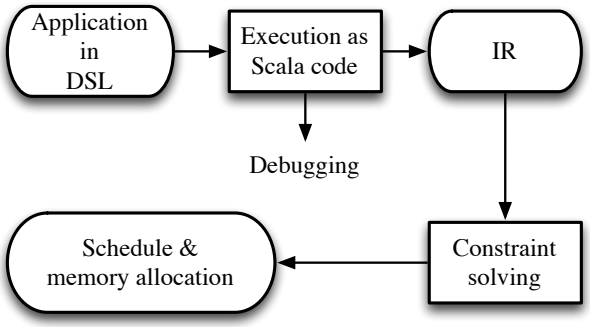


Figure 2: Programming support flow

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.

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.

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.

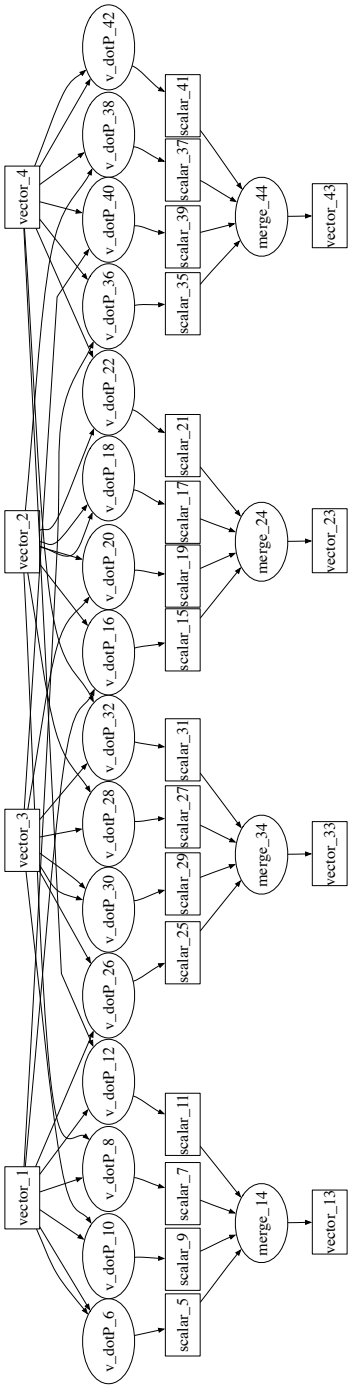


Figure 3: Intermediate representation of listing 1

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:_*)
```

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).

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 5). In some of the cases it is possible to represent a matrix operation as four vector operations, each resulting in a scalar output. Fig. 4 depicts such a vector implementation of the matrix operation in Figure 5. However, the scalar outputs should then be merged to form the vector re-

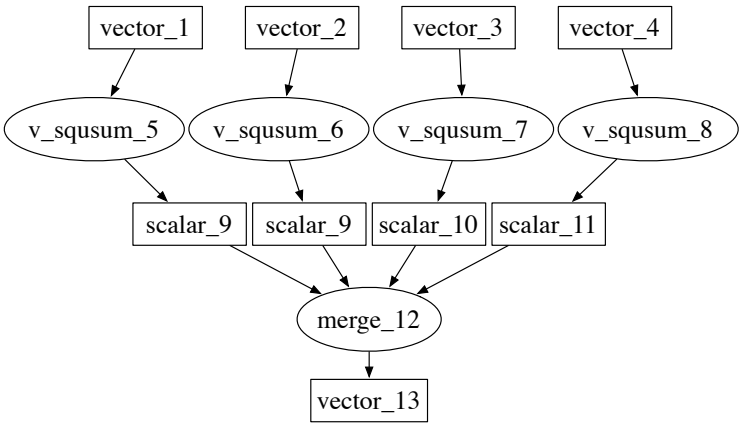


Figure 4: Vector implementation of `A.m_squsum` in Figure 5

sult, 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.

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.

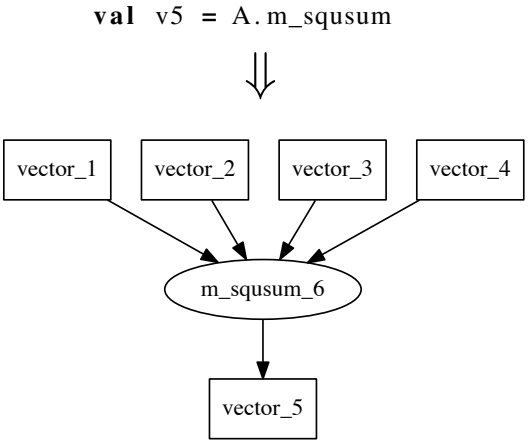


Figure 5: A matrix operation in the DSL an its IR

3.2.3 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 and Figure 7. 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.

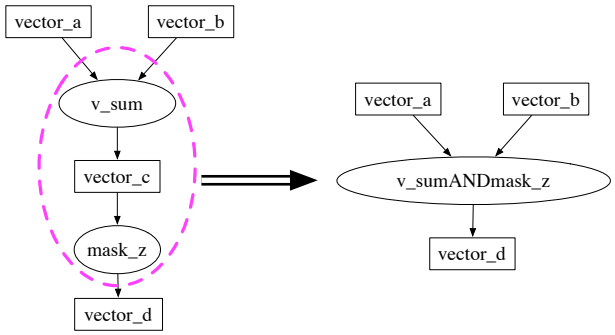


Figure 6: Merging a vector operation with a pre-processing operation.

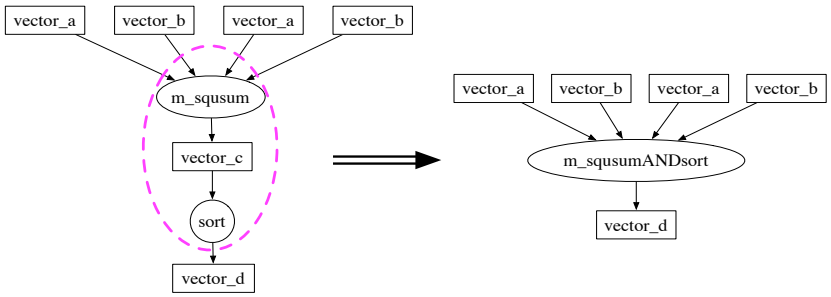


Figure 7: Merging a matrix operation with a post-processing operation when the post-processing is done on the vector output.

3.3 Scheduling an application

3.3.1 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 3.1).

For the group of operations 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)).

$$\begin{aligned}
\forall (i,j) \mid cat(i) = cat(j) = vector_op \wedge op(i) \neq op(j):\\
s_i \neq s_j
\end{aligned} \tag{3}$$

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.

3.3.2 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.

$$\begin{aligned}
\forall i \mid cat(i) = vector_data \vee cat(i) = scalar_data:\\
s_i = s_{pred(i)} + l_{pred(i)}
\end{aligned} \tag{4}$$

3.3.3 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 8), 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 9. 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.

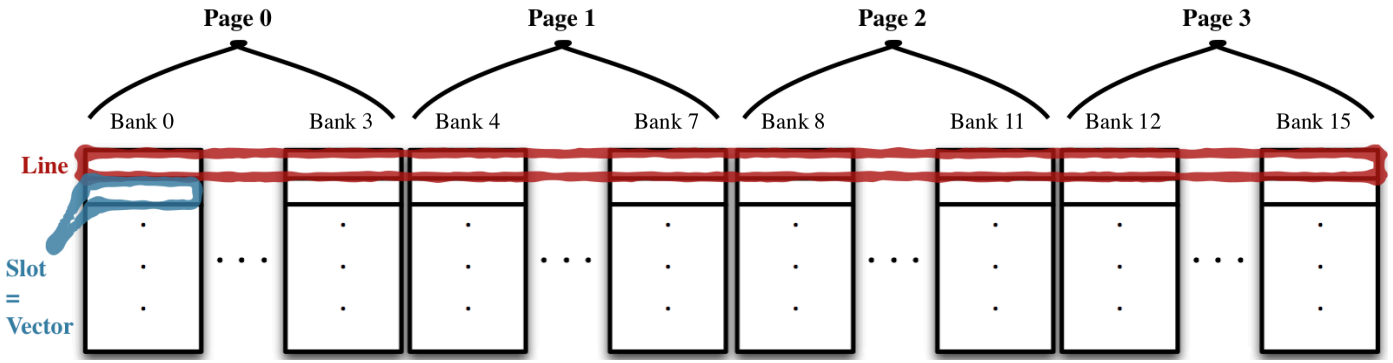


Figure 8: Memory layout abstraction. Memory is organized in pages, lines and slots.

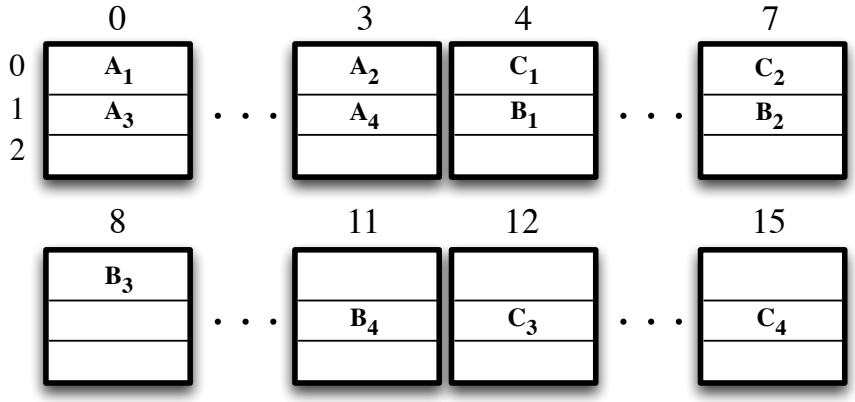


Figure 9: Memory access examples. Only *C* can be accessed in 1 cycle

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.

$$\begin{aligned} &\forall i \mid cat(i) = vector\_data : \\ &line_i = slot_i / nOfBanks \\ &page_i = (slot_i \mod nOfBanks) / pageSize \end{aligned} \tag{6}$$

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).

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

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)).

$$\begin{aligned}
&\forall i,j \mid cat(i) = cat(j) = vector_op \ \wedge \ s_i = s_j : \\
&\quad \forall (d,e) \mid d \in pred(i) \ \wedge \ e \in pred(j) \ \wedge \\
&\quad\quad cat(d) = cat(e) = vector_data : \\
&\quad page_d = page_e \implies line_d = line_e
\end{aligned} \tag{8}$$

$$\begin{aligned}
&\forall i,j \mid cat(i) = cat(j) = vector_op \ \wedge \ s_i = s_j : \\
&\quad \forall (d,e) \mid d \in succ(i) \ \wedge \ e \in succ(j) \ \wedge \\
&\quad\quad cat(d) = cat(e) = vector_data : \\
&\quad page_d = page_e \implies line_d = line_e
\end{aligned} \tag{9}$$

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.

$$\begin{aligned}
&\forall i \mid cat(i) = vector_data : \\
&\quad life_i = \max U_i - s_i \\
&\quad where : \\
&\quad U_i = [s_o \mid o \in succ(i)]
\end{aligned} \tag{10}$$

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 3.1 as seen in constraint (11).

$$\begin{aligned}
&\texttt{Diff2}(S,\ SL,\ L,\ ones) \\
&where : \\
&S = [s_i \mid cat(i) = vector_data], \\
&SL = [slot_i \mid cat(i) = vector_data], \\
&L = [life_i \mid cat(i) = vector_data]
\end{aligned} \tag{11}$$

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 3.1), 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.

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

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

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, 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

# 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

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.

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

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 trade-off 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.

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.

References

[1] Chenxin Zhang. "Dynamically Reconfigurable Architectures for Real-time Baseband Processing". PhD thesis. Lund University, 2014. URL: http://lup.lub.lu.se/record/4406448/file/4406451.pdf.

[2] Chenxin Zhang, Liang Liu, and Viktor 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] Krzysztof Kuchcinski. "Constraints-Driven Scheduling and Resource Assignment". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 8.3 (July 2003), pp. 355–383.

[4] Willem-Jan van Hoeve and Irit Katriel. "Handbook of Constraint Programming". In: Foundations of Artificial Intelligence. Elsevier Science, 2006. Chap. Global Constraints.

[5] Abderrahmane Aggoun and Nicolas Beldiceanu. "Extending chip in order to solve complex scheduling and placement problems". In: *Mathematical and Computer Modelling* 17.7 (1993), pp. 57–73.

[6] Nicolas Beldiceanu and Evelyne Contejean. "Introducing global constraints in CHIP". In: *Journal of Mathematical and Computer Modelling* 20.12 (1994), pp. 97–123.

[7] Chris Lattner and Vikram 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] Gabriel Santin Hjort Blindell. *Survey on Instruction Selection: An Extensive and Modern Literature Study*. Tech. rep. ISBN: 978-91-7501-898-0. Stockholm, Sweden: KTH Royal Institute of Technology, Oct. 2013.

[9] Andrzej Bednarski and Christoph Kessler. "Integer Linear Programming versus Dynamic Programming for Optimal Integrated VLIW Code Generation". In: *12th Int. Workshop on Compilers for Parallel Computers*. 2006.

[10] Robert Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. "Combinatorial Spill Code Optimization and Ultimate Coalescing". In: *ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES' 14. Edinburgh, UK, June 2014.

[11] Abid M. Malik, Jim McInnes, and Peter van Beek. *Optimal basic block instruction scheduling for multiple-issue processors using constraint programming*. Tech. rep. In: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence, 2005.

[12] Mattias V. Eriksson and Christoph W. Kessler. “Integrated Modulo Scheduling for Clustered VLIW Architectures”. English. In: *High Performance Embedded Architectures and Compilers*. Ed. by André Seznec, Joel Emer, Michael O’Boyle, Margaret Martonosi, and Theo Ungerer. Vol. 5409. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 65–79.

[13] Monica Lam. “Software Pipelining: An Effective Scheduling Technique for VLIW Machines”. In: *SIGPLAN Not.* 23.7 (June 1988), pp. 318–328.

[14] Grigorios Dimitroulakos, Stavros Georgiopoulos, Michalis D. Galanis, and Costas E. Goutis. “Resource Aware Mapping on Coarse Grained Reconfigurable Arrays”. In: *Microprocess. Microsyst.* 33.2 (Mar. 2009), pp. 91–105.

[15] Mehmet Ali Arslan and Krzysztof Kuchcinski. “Instruction selection and scheduling for DSP kernels”. In: *Microprocessors and Microsystems* 38.8, Part A (2014), pp. 803–813. URL: `http://www.sciencedirect.com/science/article/pii/S0141933114000520`.

[16] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. 1st. USA: Artima Incorporation, 2008.

[17] Peter Luethi, Andreas Burg, Simon Haene, David Perels, Norbert Felber, and Wolfgang 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] Mehmet Ali Arslan, Flavius Gruian, and Krzysztof Kuchcinski. “A comparative study of scheduling techniques for multimedia applications on SIMD pipelines”. In: *Proceedings of the DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS 2015)*. 2015. URL: `http://arxiv.org/pdf/1502.07447.pdf`.

[19] John Ruttenberg, Guang R. Gao, Artour Stoutchinin, and Woody 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*. PLDI ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 1–11.

A Comparative Study of Scheduling Techniques for Multimedia Applications on SIMD Pipelines

PAPER III

Abstract

Parallel architectures are essential in order to take advantage of the parallelism inherent in streaming applications. One particular branch of these employ hardware SIMD pipelines. In this paper, we analyse several scheduling techniques, namely ad hoc overlapped execution, modulo scheduling and modulo scheduling with unrolling, all of which aim to efficiently utilize the special architecture design. Our investigation focuses on improving throughput while analysing other metrics that are important for streaming applications, such as register pressure, buffer sizes and code size. Through experiments conducted on several media benchmarks, we present and discuss trade-offs involved when selecting any one of these scheduling techniques.

Mehmet Ali Arslan, Flavius Gruian and Krzysztof Kuchcinski
Lund University, Department of Computer Science

*Based on the paper published in the proceedings for DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS 2015), 2015-03-13, Grenoble, France, 2015

1 Introduction

With the growing demands on mobile computing, fuelled among others by the advent of the Internet of Things (IoT) and cloud computing, the amount of processing required from multimedia and telecommunication applications are also increasing. Such applications typically process streams of data, exhibiting a high degree of parallelism, requiring high performance with a limited power budget. The answer, from the hardware point of view, is the introduction of highly parallel architectures.

Parallelism can be achieved at different levels in an architecture, by employing pipelining, SIMD processing, multi-core or array processors. The current trend in commercial system is to use multi-core systems extended with graphic processors (GPU). In mobile devices the alternative is often offered by using DSP processors and ASIC accelerators. Furthermore, some applications require custom architectures, designed especially to provide high computational power, for very specific programming models.

Combining all these techniques and parallelism at different levels can in principle yield architectures with enough computational power and low power consumption, but their programming is not trivial. The main challenge today is therefore programming these platforms such that the applications fully utilize the hardware resources. Furthermore, different implementation choices offer different trade-offs in terms of performance, memory consumption, power, etc. Therefore, depending on the application context, it is important to carefully consider alternatives rather than committing to the "best" implementation.

In this paper, we analyse and compare different ways of scheduling repetitive behavior (kernels) when compiling code for parallel architectures. In particular, the target architecture we focus on is a generic architecture that employs a SIMD pipeline. This architecture model is abstract enough to model a class of architectures, including a custom reconfigurable architecture designed for our specific application area [1]. That architecture centres around a highly reconfigurable pipelined processor with vector instructions (SIMD). Reconfigurations may be carried out in one clock cycle, which makes for a very flexible instruction set, offering a large number of execution and optimization choices. Furthermore, the architecture model we adopt in this paper is also rather similar to the GPU hardware.

The remainder of the paper is organized as follows. Section 2 briefly presents the previously published work in the context of scheduling for pipelines and SIMD architectures. Section 3 presents the generic architecture model as well as other assumptions adopted in the paper. Section 4 describes the scheduling techniques studied in this paper along with a brief overview of the method we used to obtain such schedules. Section 6 presents the experimental setup and results while section 6 employs these results to compare the techniques described earlier. Finally, our conclusions are drawn in section 7.

2 Related Work

Optimal use of the instruction level parallelism (ILP) through software pipelining [2] has been addressed in the literature early on. One of the standard techniques, first proposed in [3], is *modulo scheduling*, which selects a minimal schedule for one loop iteration such that no constraints are violated when the schedule repeats.

Combined with retiming and unrolling, modulo scheduling can be even more effective, as shown in [4, 5, 6]. Nevertheless, such techniques, intended to increase the instruction per clock (IPC) count or throughput, suffer instead from a drastic increase in live data, thus register requirements. Therefore, much of the work also focused on minimizing the register pressure [7, 8, 9] while scheduling.

Computing architectures with high ILP that benefit from software pipelining are today ubiquitous. One of the earliest uses of the technique is described in [10] which targets very large instruction word (VLIW) architectures. Vectorization for single instruction, multiple data (SIMD) architectures has also been tackled early on [11, 12, 13], while more recent work even targets streaming applications [14]. Software pipelining and especially modulo scheduling was also proven to be successful for newer architectures, designed for offering a high degree of parallelism, namely coarse-grained reconfigurable arrays (CGRA) [15, 16, 17]. Even more exotic architectures, especially designed for streaming applications have been shown to benefit from software pipelining [1, 18].

In this paper we revisit modulo scheduling along with partially unrolled modulo scheduling, as well as another common but ad hoc practice we call *overlapped execution* (see section 4.2). However, the focus of our investigation is on those measures that are relevant for streaming applications, namely throughput and input/output data rates. Register pressure (minimal register requirements) as well as code size are also examined, in order to underline the differences between the various techniques. The interaction between registers, extent of unrolling and code size has been studied previously in [19]. In that work, however, the focus is on VLIW processors, and unrolling is basically carried out after scheduling, while we target SIMD processors, and unroll before scheduling.

For streaming applications, using modulo scheduling in order to increase throughput, while also keeping buffer sizes under control, has already been addressed in the literature [20, 21, 22]. However, it is important to notice that the type of pipelining targeted in these approaches is *algorithmic pipelining* on multi/many-cores. In that case scheduling employs processor level parallelism (PLP), rather than ILP as in our case. By replicating tasks (*actors*) and groups of actors on the same or several processors, data production and consumption rates are negotiated in such a way that buffers sizes are minimized. Note that each instance of an actor execution is treated as an atomic behaviour with a given latency. The type of scheduling we address in this paper is complementary to the algorithmic pipelining used in those approaches, adding another degree of freedom to the design flow. Instead of assuming a fixed atomic execution (yet repetitive) of an actor, we show

that its behaviour can be pipelined in different ways, yielding different input/output rate which will affect the choices of algorithmic pipelining.

An approach that does use both ILP and PLP for streaming application is described in [14], employing SIMDization at several levels. However, their use of ILP is restricted to stateless actors, whose instances are combined to exactly cover the vector size and execute in parallel. A similar technique for loop SIMDization, focused on memory access patterns, is presented in [13]. In contrast, our use of ILP allows for more generic architectures, execution models, and actors.

3 Context

As target applications, we consider kernels from digital signal processing (DSP), which are part of image processing algorithms. Such kernels can be implemented differently depending of the computational model. For example, in an imperative language they are usually implemented as an iterative execution of a code sequence (a loop or a nested loop construct). In the dataflow model of computation [23], this is equivalent to the repetitive execution of an actor processing a data stream.

For this study, we confine ourselves to kernels that have no inter-iteration dependency, which allows us to emphasize the differences between various execution scenarios. However, our scheduling methods are not limited by this assumption and can be extended with additional constraints modelling inter-iteration dependencies.

In the following, we use benchmarks that represent inner loop bodies or actor computations. Each benchmark is modelled as a *directed acyclic graph* (DAG) whose nodes represent basic operations and edges dependencies between them. Similar basic operations, which are also independent, can thus be grouped together and issued as one SIMD instruction.

We consider a generic target architecture with a hardware pipeline that can perform SIMD operations of a given width. The hardware pipeline has a given number of stages, which defines the latency between data dependent operations, since we assume that data must be available at pipeline start and produced by the last stage. Furthermore, we assume that each pipeline stage has a latency of one clock cycle. This particular choice of the architecture is motivated by our previous work with the system in [1].

4 Approach

We compare different scheduling techniques for DSP kernels, usually employed to increase the throughput. The measures we examine for each technique are those we consider relevant for streaming applications, namely throughput, register pressure, input/output data rates, and code size. We also investigate how different architecture configurations influence these measures.

The techniques we consider are: scheduling a single iteration, overlapping iterations, along with modulo scheduling, classic as well as combined with loop unrolling. The experiments are carried out on several graphs from ExpressDFG [24] which provides dataflow graphs for benchmark applications from the MediaBench suite [25].

Practically the tools implementing the different scheduling methods were developed using the constraint programming environment offered by JaCoP [26]. The modelling of these methods using constraint programming is itself an interesting problem, which we addressed in detail in our previous work. In the following, without going into the modelling issues, we briefly present the scheduling techniques we investigate in this paper.

4.1 Scheduling one iteration

The straightforward way to schedule a loop/kernel is to sequentially execute iterations, which requires scheduling one iteration efficiently. To illustrate, we will use a small part of an elliptic wave filter (EWF), taken from ExpressDFG [24]. Figure 1 shows the shortest schedule, given the assumptions described in section 3.

In the figure, the DAG format is preserved to show the dependencies and schedule time is shown on the horizontal axis. We deliberately picked this part from EWF to illustrate the scheduling behavior when the available ILP is limited and the critical path dominates the schedule. In this example, the resource utilization, calculated by the number of operations that are scheduled, over the number of nodes the architecture can run during the schedule length, is around 6%. Throughput is 0.024 samples per clock cycle and latency is 42 clock cycles. This poor utilization is caused both by the properties of the architecture and the application, as we detail in the following.

The long latency of the hardware pipeline (which is equal to 7 for this example - see section 3) is one reason. Due to the data dependencies (mainly through the critical path), each dependent node has to wait for its predecessor to finish its execution, i.e. go through all the pipeline stages.

The low ILP present in the application is the other reason for the poor resource utilization. Note from the figure that some of the nodes are scheduled simultaneously. This is enabled by the SIMD nature of the architecture which execute up to SIMD width (4 in this example) number of operations. However, in this case there are not enough operations that are mutually independent, hence not many can be scheduled together.

Poor utilization generally means poor throughput and low energy efficiency. To increase both, it is common procedure to schedule several iterations simultaneously.

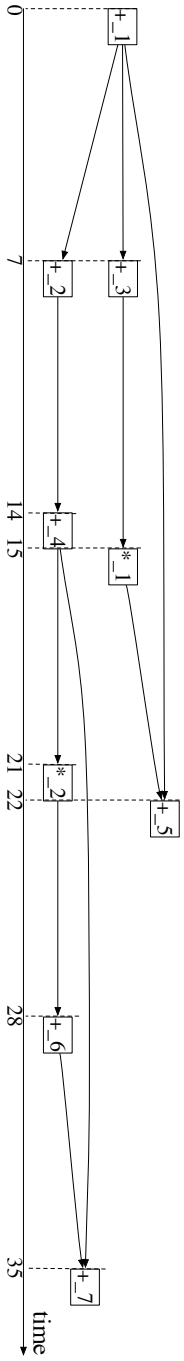


Figure 1: Example schedule of a single iteration.

4.2 Overlapping execution

For architectures with a hardware pipeline, an easy way to increase utilization is to schedule multiple iterations in an ad hoc, overlapping fashion. The scheduling process is two-fold. First the instructions for a single iteration are selected and ordered, with the objective of minimizing the number of effective (non-*nop*) instructions (in contrast to schedule length). The overlapped schedule is obtained then by advancing the chosen number of iterations by running the same corresponding instruction from each one in sequence. Once all equivalent instructions from all iterations have been scheduled, the execution advances to the next instruction. The resulting schedule for the example from Figure 1 is given in Figure 2, where iterations are denoted with upper case letters in the node identifier.

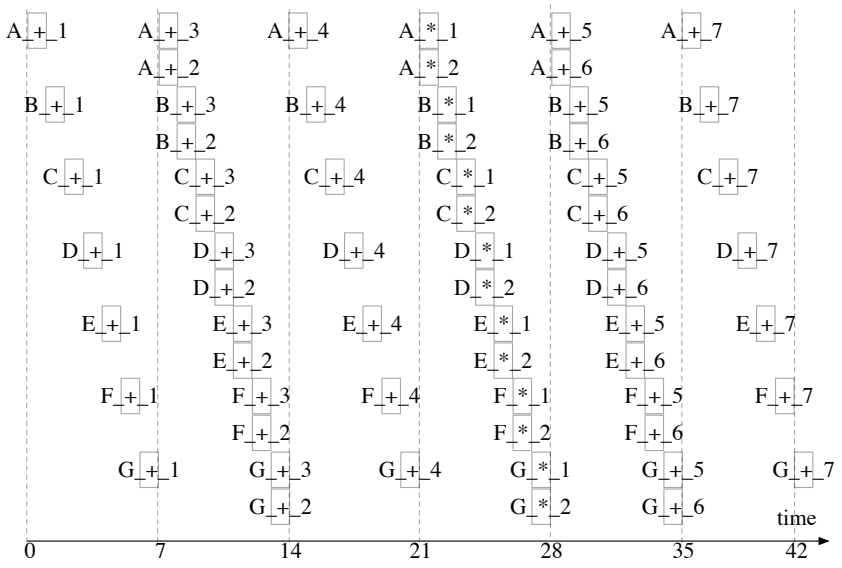


Figure 2: Example schedule of an overlapped execution.

This computationally simple solution eliminates the gaps in the schedule caused by the hardware pipeline completely, as long as the number of iterations scheduled simultaneously is larger than the pipeline latency/length. For this example the utilization is increased to 32%. The throughput is 0.146 samples per clock cycle and latency is 42 clock cycles.

Besides its simplicity, this approach is suitable particularly for certain reconfigurable architectures for which the reconfiguration overhead is a significant issue in scheduling. A reconfiguration is needed when two different types of instructions follow each other, and this approach limits it to the number of instructions. (More details for this approach and its usage can be found in [27])

4.3 Modulo scheduling

As mentioned in section 2, *modulo scheduling* is a common technique for increasing the throughput of a kernel. It involves finding a schedule that initiates iterations as soon as possible, taking into account dependencies and resource constraints, while also repeating regularly with a given interval (*initiation interval II* [28]).

Scheduling the example from Figure 1 gives an II of length 3, as depicted in Figure 3a. Nodes from the same iteration are coded with the same upper case letter, in the same order as the iterations, to give an idea about how the II is assembled from different iterations. With this notation, node A_+_7 is the last operation from iteration A, while node M_+_1 is the first operation from the latest iteration about to start executing. Note that iteration M is twelve iterations later than A. The iterations in between B and L have already started in previous IIs and are currently executing (are active) in the pipeline. It is interesting to note however that not all iterations have operations that start in every II instance, although they may be in the process of executing operations or simply wait for data. With our notation, iterations B, D, G, J, L, which correspond to iterations with distances of $1, 3, 6, 9, 11$ relative to A, are *invisible* since they are not starting any operations in the II depicted in Figure 3a. Note however that from an absolute point of view, the set of *invisible* iterations will change with the II instances. The existence of these iterations is an artefact of the short II relative to the rather long hardware pipeline.

For the given SIMD width of four, only the first clock cycle in Figure 3a utilizes the hardware fully. Elsewhere the hardware is underutilized because of insufficient ILP in the application, even using modulo scheduling. The average utilization in the II is 75%, while the throughput is 0.33 samples per clock cycle and latency is 44 clock cycles.

Compared to the overlapped execution, modulo scheduling increases the ILP, to some extent, both by folding the graph as well as filling in the gaps resulted from the pipeline latency.

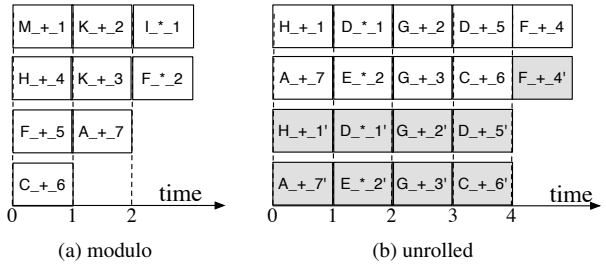


Figure 3: Initiation intervals for modulo and unrolled modulo scheduling, for the example in Figure 1.

4.4 Unrolling and modulo scheduling

Even though modulo scheduling increases the available ILP by folding several iterations to construct the II, it can be seen from Figure 3a that there are still slots in the schedule that are not utilized. To remedy this, an idea is to unroll several iterations prior to modulo scheduling. This way, the available ILP can be increased further.

In Figure 3b we depict the II for our running example when two iterations are unrolled before modulo scheduling. In addition to the notation from Figure 3a, we use different shading to distinguish between nodes unrolled from different iterations. For example, H_+_1 and H_+_1' refer to the first node from the first and the second unrolled instances, respectively, which together constitute iteration H. With the increased number of independent nodes, this method makes better use of the SIMD capability compared to classic modulo scheduling. Thus the utilization increases to 90% for this example, throughput is increased to 0.40 samples per clock cycle while the latency becomes 42 clock cycles.

5 Experiments

In this section, we present our experiments for measuring the quality of the schedules provided by the methods introduced earlier. A deeper discussion around these results makes the subject of the following section. For these experiments, unless stated otherwise, we assumed that target architecture, introduced in section 3, has a pipeline length of seven stages, and a SIMD width of four. These parameters are varied occasionally as stated, in order to investigate their impact on the measures under scrutiny.

To illustrate various characteristics of the scheduling techniques, we use Loeffler's IDCT [29] (referred to as *IDCT* in the rest of the paper) to carry out a more detailed analysis. IDCT is part of many image and video applications including JPEG and MPEG, and Loeffler's IDCT is a commonly used implementation. Note however, that similar results are observed for the other kernels we experimented with (see Tab. 2).

A summary of the results for scheduling IDCT is given in Tab. 1. Number of nodes and edges in the graph are denoted with $|V|$ and $|E|$, respectively. Each column refers to a different scheduling technique. The column "single" refers to the results for scheduling one iteration and constitutes a baseline for comparison. For data related metrics, one iteration is assumed to consume and produce one unit of data, referred to as a *sample* in the table. Overlapped scheduling interleaves seven iterations, since the pipeline length/latency is seven. Unrolled modulo scheduling, referred to as *unrolled* in the rest, involves unrolling two iterations.

For single and overlapped scheduling, the *throughput* is calculated as $\#iterations/schedulelength$. When computing this for modulo and unrolled the $schedulelength$ is replaced by II, since at the end of each II, exactly $\#iterations$

finish executing and produce their outputs. The *latency* for an iteration is the distance between its first scheduled input node and last scheduled output node. The latency of all iterations in an overlapped schedule is the same since each iteration is only a shifted version of the first. In contrast, as the unrolled iterations are scheduled freely to obtain a minimal II, their latencies may differ.

IDCT ($\|V\| = 48, \|E\| = 63$)	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.022	0.046	0.059	0.080
Latency (cc)	46	147	53	67, 68
Code size (instr.)	46	153	17	25

Table 1: Loeffler's IDCT with various scheduling techniques.

Code size denotes the number of instructions that would be generated from the schedule. This includes the possible no operation (*nop*) instructions. For single and overlapped scheduling this figure is equal to the schedule length. For modulo and unrolled, the prologue and epilogue are negligible when the kernel is run many times, thus the code size for these methods is equal to their II.

Assuming that live data is kept in registers, the number of registers needed throughout each schedule is plotted in Fig. 4. Computing the lifetimes for input/output data is interesting, since this is directly coupled to the buffer sizes, when it comes to streaming applications. Input data is considered alive from the first operation using (part of) it, while output data is alive until the last operation producing (part of) it. Input data becoming alive is reflected through the consumption of an input sample from the input buffers. Similarly for output data end of life and sample production in the output buffers.

Note again, that the overlapped schedule comprises seven iterations. Modulo and unrolled schedules are plotted only over one II, since this profile repeats every II. Here *unrolled x2* unrolls two iterations and *unrolled x5* unrolls five.

Throughput is a performance metric that often refers to the average output of the schedule. However, for some application domains, such as streaming, the instant rates of input consumption and output production is more important, also affecting the buffer sizes. To show the variation in these rates between the various scheduling techniques, Fig. 5 plots the cumulative input consumption and output production for IDCT, over a longer time period.

Table 2 summarizes the results for scheduling three additional kernels, taken from ExpressDFG [24]. Number of nodes and edges for each graph is denoted with $|V|$ and $|E|$, respectively, under the application name. The last kernel in Table 2 is a different IDCT implemented in MPEG, which should not be confused with Loeffler's IDCT.

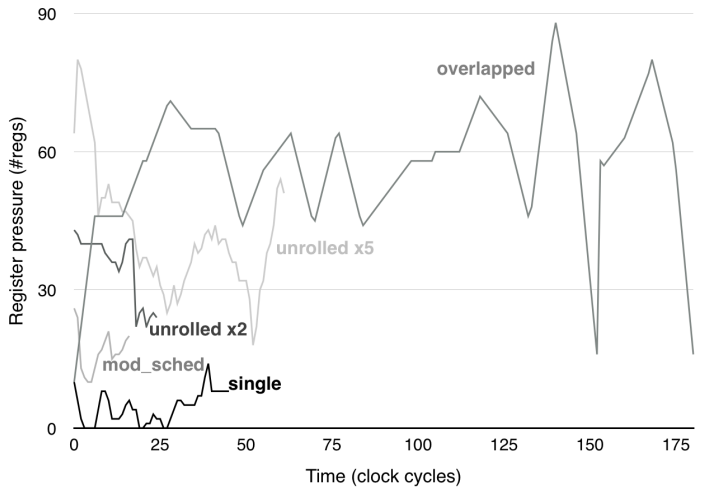


Figure 4: Register pressures for IDCT scheduled with various techniques. Each plot covers one execution of the respective schedule.

EWF ($\|V\| = 34, \|E\| = 47$)	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.010	0.067	0.111	0.118
Latency (cc)	98	98	104	(98, 98)
max. reg. press (registers)	9	41	49	52
data cons. (samples)	1	7	1	1
data prod. (samples)	1	7	1	1
Code size (instr.)	98	104	9	17
JPEG FDCT ($\|V\| = 134, \|E\| = 169$)	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.011	0.020	0.025	0.026
Latency (cc)	92	343	122	(157, 155)
max. reg. press (registers)	26	198	64	62
data cons. (samples)	1	7	1	2
data prod. (samples)	1	7	1	1
Code size (instr.)	92	349	40	78
MPEG IDCT ($\|V\| = 114, \|E\| = 164$)	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.009	0.016	0.021	0.023
Latency (cc)	115	420	138	(143, 143)
max. reg. press (registers)	26	204	48	68
data cons. (samples)	1	7	1	1
data prod. (samples)	1	7	1	1
Code size (instr.)	115	426	48	88

Table 2: Performance measures for three benchmarks scheduled with various techniques.

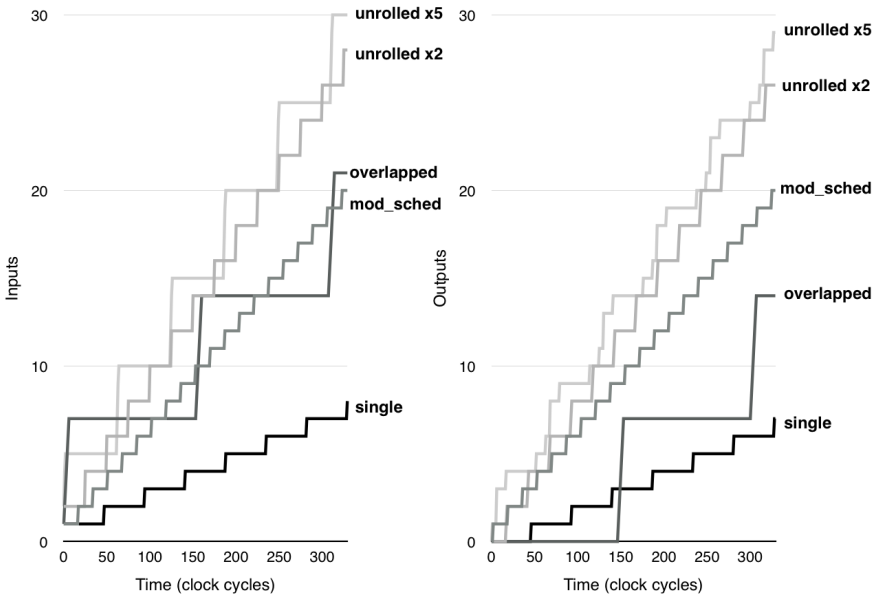


Figure 5: Input consumption and output production for IDCT scheduled with various techniques.

Apart from the metrics in Tab. 1, there are three additional metrics that summarize the register pressure, input and output rates for these kernels. (For IDCT, Fig. 4 and 5 present this information in more detail). The metric "max. reg. press" denotes the maximum register pressure in the schedule. To present the input/output behavior, we also use two other metrics, namely "data cons." and "data prod." Both use a time window that slides through the schedule and captures the maximum sample consumption and production, respectively. The length of this sliding window is set to the number of pipeline stages, in order to better capture the behavior of the overlapped schedule.

In order to show the differences between different extents of unrolling, we include *unrolled x5* and *unrolled x2* in both Fig. 4 and Fig. 5.

Further results from experiments with different extent of unrolling are given in Table 3. In addition to previously mentioned metrics, we also include resource utilization. This is computed as $n/(II*N)$ where n denotes the number of nodes in the graph and N denotes the SIMD width (i.e. the number of operations that can be run simultaneously). We include utilization figures in order to give an idea of the efficiency of the schedule and the untapped parallelism still existent in the architecture. The "Latencies" row shows that the different iterations unrolled and scheduled together may in fact get different latencies. The reason is that latency is not in any way constrained in our CP model for modulo scheduling. If the

application domain requires the latency for each sample output to be equal (or close to equal) this can be included in the model. However, it should be noted that the resulting *II* can degrade, since the solution space becomes more constrained.

Besides the target application and the scheduling method, there are two parameters of the architecture that can affect the schedules, namely the SIMD width and the number of pipeline stages. We varied both parameters and gathered the results for IDCT in Tab. 4 and 5, respectively.

SIMD width = 8	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.022	0.053	0.143	0.143
Latency (cc)	46	126	53	57, 57
max. reg. press (registers)	12	88	47	61
data cons. (samples)	1	7	1	1
data prod. (samples)	1	7	1	1
Code size (instr.)	46	132	7	14
SIMD width = 16	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.022	0.072	0.250	0.286
Latency (cc)	46	97	52	53, 53
max. reg. press (registers)	14	88	73	93
data cons. (samples)	1	7	2	1
data prod. (samples)	1	7	2	1
Code size (instr.)	46	97	4	7

Table 4: The effect of SIMD width on IDCT schedules.

# iterations unrolled	**two**	**three**	**four**	**five**
Throughput (samples/cc)	0.080	0.075	0.083	0.081
Utilization (n/(II*N))(%)	96	90	100	97
Latencies (cc)	67,68	79, 85, 82	52, 97, 96, 91	63, 68, 78, 113, 66
max. reg. press (registers)	43	57	66	80
data cons. (samples)	1	2	2	3
data prod. (samples)	2	3	2	2
Code size (instr.)	25	40	48	62

Table 3: The effect of changing the number of iterations unrolled for IDCT.

# Pipe. Stages = 4	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.037	0.051	0.067	0.080
Latency (cc)	27	76	30	49, 49
max. reg. press (registers)	16	52	18	35
data cons. (samples)	1	4	1	1
data prod. (samples)	1	4	1	1
Code size (instr.)	27	79	15	25
# Pipe. Stages = 16	**Single**	**Overlapped**	**Modulo**	**Unrolled x2**
Throughput (samples/cc)	0.010	0.046	0.071	0.077
Latency (cc)	100	336	102	113, 115
max. reg. press (registers)	14	196	35	48
data cons. (samples)	1	16	2	1
data prod. (samples)	1	16	2	2
Code size (instr.)	100	351	14	26

Table 5: The effect of pipeline length on IDCT schedules.

6 Discussion

It is obvious from the previous section that the different scheduling techniques offer different trade-offs between the performance metrics we consider. Depending on the context, the choice of one or another technique may be preferred. However, we can claim with high confidence that scheduling a single iteration results in overall poor performance.

At a first glance the overlapped execution offers advantages only over the single iteration schedule, in terms of throughput. However, there are some hidden advantages that should be considered. First, obtaining an overlapped schedule is easier than obtaining modulo and unrolled schedules, which require specific techniques. In fact starting from a single iteration schedule, obtaining the overlapped schedule is straightforward. Second, overlapped execution may in fact be more effective on selected reconfigurable architectures, designed for streaming applications [27]. To keep our architecture abstraction simple, we did not include reconfigurability in this work. However in reconfigurable architectures where reconfiguration takes a certain time (e.g. one clock cycle) to switch between different instructions, overlapped can outperform modulo scheduling in terms of throughput. We present further insights into the effects of reconfiguration for overlapped and modulo scheduling in [27].

6.1 Average throughput

In all examples, modulo and unrolled provide significant improvement in throughput over the overlapped scheduling, which is itself an improvement from single iteration scheduling.

The number of iterations that are unrolled affects the throughput as well. This is captured in Table 3. For IDCT, unrolling four iterations gives the best throughput since it reaches full utilization. Further unrolling is in principle not needed and introduces mismatches between the application ILP and the architecture. Obviously for a different application, the optimum number of unrolls will be different. It should be noted as well, that increasing the number of unrolled iterations increases in turn register pressure and code size, and possibly also the required buffer sizes. Additionally it may also incur latency penalties for some of the iterations that are unrolled.

Looking at the architecture impact on throughput, a wider SIMD architecture produces a better throughput, but only when the application ILP allows it (see Table 4). This also means that more unrolling, which increases ILP also may better employ the architecture, thus increasing throughput. On the other hand, the pipeline length seems to have only marginal effect on the throughput for all techniques except, obviously, the single iteration schedule (see Table 5).

6.2 Code size

Modulo scheduling techniques fold the schedule into the *initiation interval*, that eventually is a fraction of a regular sequential schedule. As a result of this, modulo and unrolled schedule yield smaller code size compared to the overlapped execution, which basically includes all the iterations that are overlapped. Single iteration schedules are rather long in terms of code size, but are also sparse, including a number of *nop*s introduced due to dependencies and pipeline latency. Overall, modulo scheduling (which is equivalent to *unrolled x1*) is the most efficient when it comes to code size.

From an architectural point of view, wider SIMD units means also shorter code (see Table 4), when enough ILP is available in the application, since more work can be done in the same clock cycle. In contrast, longer pipelines seem to have again little influence on the code size for modulo and unrolled execution, but greatly affect the code size for overlapped and single iteration schedules. This is not unexpected, since the pipeline length directly dictates the schedule length for both of the latter, meanwhile modulo scheduling manages to wrap around (several times possibly) the length of the pipeline, hiding its latency.

6.3 Storage requirements and data rates

Register pressure is decisive for architectures with small register files and slow memory. From Fig. 4 and all the tables, it is apparent that single execution always needs the least amount of registers, while overlapped execution uses the most, with modulo scheduling being somewhere in between. The amount of unrolling for modulo scheduling also directly affects the register pressure. Increasing the SIMD width, allows for more computations in parallel, thus increasing the pressure on the registers (see Table 4). Longer pipelines also seem to increase the register pressure, since data is alive over longer time intervals (see Table 5).

When it comes to dimensioning buffers in streaming applications, input/output data rates are of paramount importance. Buffer sizes need to be selected in such a way that they never stall computations upstream or downstream due to lack of space or data. Opting for a high throughput implementation when the overall application context cannot support it would be a bad design choice. In here we refer to the distance between two consecutive inputs/outputs as *burstiness*. From Figure 5, it is evident that overlapped execution is the most bursty scheduling technique while modulo is the least bursty. This effect is accentuated for longer pipelines. We note also that unrolling appears to increase the burstiness and the irregularity of input/output rates for modulo scheduling, proportionally to the number of iterations unrolled.

7 Conclusions

In this paper, we experimentally analysed three different scheduling techniques to compare their performance with respect to throughput, latency, register/memory pressure, data consumption/production rates and code size. We identified different bottlenecks caused by the application and the architecture. We then presented different trade-offs that are to be taken into account when employing one of the scheduling techniques, to compile a streaming application kernel on a generic parallel architecture that employs a SIMD hardware pipeline. Furthermore, we investigated the effects of altering the parameters of the generic architecture, namely the number of pipeline stages and SIMD width.

References

[1] Chenxin Zhang. "Dynamically Reconfigurable Architectures for Real-time Baseband Processing". PhD thesis. Lund University, 2014.

[2] Janak H. Patel and Edward S. Davidson. "Improving the throughput of a pipeline by insertion of delays". In: *ACM SIGARCH Computer Architecture News*. Vol. 4. 4. ACM. 1976, pp. 159–164.

[3] B. Ramakrishna Rau and Christopher D. Glaeser. "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing". In: *ACM SIGMICRO Newsletter*. Vol. 12. 4. IEEE Press. 1981, pp. 183–198.

[4] Keshab K. Parhi and David G. Messerschmitt. "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding". In: *Computers, IEEE Transactions on* 40.2 (1991), pp. 178–195.

[5] Daniel M. Lavery and Wen-Mei W. Hwu. "Unrolling-based optimizations for modulo scheduling". In: *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE Computer Society Press. 1995, pp. 327–337.

[6] Alain Darte and Guillaume Huard. "Loop shifting for loop compaction". In: *International Journal of Parallel Programming* 28.5 (2000), pp. 499–534.

[7] B. Ramakrishna Rau, Meng Lee, Parthasarathy P. Tirumalai, and Michael S. Schlansker. "Register Allocation for Software Pipelined Loops". In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. PLDI '92. San Francisco, California, USA: ACM, 1992, pp. 283–299.

[8] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. "Optimum modulo schedules for minimum register requirements". In: *Proceedings of the 9th international conference on Supercomputing*. ACM. 1995, pp. 31–40.

[9] Sid-Ahmed-Ali Touati and Christine Eisenbeis. "Early periodic register allocation on ilp processors". In: *Parallel Processing Letters* 14.02 (2004), pp. 287–313.

[10] Monica Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 318–328.

[11] John R. Allen and Ken Kennedy. *PFC: A program to convert Fortran to parallel form*. Technical Report 82–6. Department of Mathematical Sciences, Oct. 1982.

[12] Randy Allen and Ken Kennedy. "Automatic translation of Fortran programs to vector form". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.4 (1987), pp. 491–542.

[13] Dorit Nuzman, Ira Rosen, and Ayal Zaks. "Auto-vectorization of interleaved data for SIMD". In: *ACM SIGPLAN Notices*. Vol. 41. 6. ACM. 2006, pp. 132–143.

[14] Amir H. Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott A. Mahlke. "MacroSS: macro-SIMDization of streaming applications". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 1. ACM. 2010, pp. 285–296.

[15] Wonsub Kim, Yoonseo Choi, and Haewoo Park. "Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.4 (2013), p. 58.

[16] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling". In: *Computers and Digital Techniques, IEE Proceedings-*. Vol. 150. 5. IET. 2003, pp. 255–61.

[17] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 166–176.

[18] Chenxin Zhang, Liang Liu, and Viktor Öwall. "Mapping channel estimation and MIMO detection in LTE-advanced on a reconfigurable cell array". In: *IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2012, pp. 1799–1802.

[19] Mounira Bachir, Sid-Ahmed-Ali Touati, Frederic Brault, David Gregg, and Albert Cohen. "Minimal unroll factor for code generation of software pipelining". In: *International Journal of Parallel Programming* 41.1 (2013), pp. 1–58.

[20] Manjunath Kudlur and Scott A. Mahlke. "Orchestrating the execution of stream programs on multicore platforms". In: *ACM SIGPLAN Notices*. Vol. 43. 6. ACM. 2008, pp. 114–124.

[21] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott A. Mahlke, and Trevor Mudge. "Stream compilation for real-time embedded multicore systems". In: *Code generation and optimization, 2009. CGO 2009. International symposium on*. IEEE. 2009, pp. 210–220.

[22] Jongsoo Park and William J Dally. "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures". In: *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*. ACM. 2010, pp. 1–10.

[23] Edward A. Lee and Thomas M. Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (May 1995), pp. 773–801.

[24] ExpressDFG. *ExpressDFG benchmark website.* 2006. URL: `http://express.ece.ucsb.edu/benchmark`.

[25] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems". In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society. 1997, pp. 330–335.

[26] Krzysztof Kuchcinski and Radoslaw Szymanek. *JaCoP Library. User's Guide*. 2014. URL: `http://www.jacop.eu`.

[27] Mehmet Ali Arslan, Krzysztof Kuchcinski, Flavius Gruian, and Yangxurui Liu. "Programming Support for Reconfigurable Custom Vector Architectures". In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '15. San Francisco, California: ACM, 2015, pp. 49–57.

[28] B. Ramakrishna Rau. "Iterative modulo scheduling". In: *International Journal of Parallel Programming* 24.1 (1996), pp. 3–64.

[29] Christoph Loeffler, Adriaan Ligtenberg, and George S. Moschytz. "Practical fast 1-D DCT algorithms with 11 multiplications". In: *ICASSP-89, International Conference on Acoustics, Speech, and Signal Processing,* May 1989, 988–991 vol.2.

APPLICATION-SET DRIVEN EXPLORATION FOR CUSTOM PROCESSOR ARCHITECTURES

PAPER IV

Abstract

Custom processor architectures are often proposed as more efficient alternatives to general purpose processors in terms of performance and power. However, the design of such architectures requires experts both in hardware and the application domain, carrying out time consuming exploration involving detailed modeling and simulation of a number of solutions. Often such architectures are over dimensioned in order to secure the demanded performance, leading to a waste of area, power and energy. In this paper we propose a method for speeding up the design space exploration, while meeting the performance demands for a group of applications, with minimal resources. Initially our method, based on Pareto points, identifies sets of solutions in terms of scalar units and vector units of certain length, fulfilling given throughput constraints for each application in a given set. Architectures can then be selected by combining these solutions, as starting points for a more thorough, model-based evaluation. We validate our method by presenting a case study revolving around selecting an architecture suitable for a set of applications taken from the multimedia domain.

Mehmet Ali Arslan, Krzysztof Kuchcinski and Flavius Gruian
Dept. of Computer Science, Lund University

*A shorter version of this paper is published on the proceedings for 26th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2015-07-27/2015-07-29, Toronto, Canada, 2015

1 Introduction

The choice of designing a custom processor architecture is often taken as a result of identifying obstacles with existing architectures when it comes to performance, chip area, power or energy consumption, for specific application domains. Once embarked on the quest for selecting the optimal architecture for a set of applications, the designer's experience in both hardware architectures and targeted software is paramount. Given the large number of possible solutions, choosing a representative subset to investigate further is currently more of an art than a rigorous process. Subsequently these potential solutions are usually modeled in detail and used to simulate the applications in order to filter out unsuitable candidates. In extreme cases, new candidates need to be proposed, if none of the initial ones are satisfactory. Alternatively, the architecture is over-designed, employing more resources than necessary, to make sure that all of the applications meet their requirements. Neither of these cases is desirable, since they mean higher cost through a waste of resources, be it designer effort, chip area, power or energy consumption.

In this paper we propose a method for a more systematic design space exploration for application specific processors, intended to reduce the design effort in the initial phase, by proposing an optimal set of alternative potential solutions. In particular we are interested in identifying the processor configurations in terms of number and width of SIMD units and the number of scalar units that allow the required performance, without wasting resources. As the performance measure we use throughput, since we mainly target streaming applications, which are often throughput constrained. To identify potential configurations, we employ constraint programming, and formulate the problem as a Pareto optimization problem in a three dimensional space (number and width of SIMD units, number of scalar units), using modulo scheduling to ensure throughput requirements are met [1]. Once the Pareto points are found, it is up to the designer to select the ones that are more interesting to investigate further, to get more accurate estimates on cost and performance.

Note that such systematic design space exploration is not new, but was previously employed for system rather than processor level. In that case the components are fixed (processors, memory, interconnect type) and a solution refers to the number of components of different types and their interconnection. Our method could then be applied as the next phase in this design process, where the individual computing components are optimized.

2 Related Work

Design space exploration (DSE) is an important activity during embedded system design. It explores different solutions and provide trade-offs between them. The designer can then select the most appropriate system configuration. DSE is in-

tensively studied and many methods have been proposed. One can divide them into methods that use simulation, (multi-objective) optimization or combination of these. DSE is used to find design trade-offs for different design metrics.

Simulation based methods assume the existence of a model that can be executed to gather the design metrics. For this kind of simulations, existing languages and toolkits are often used, such as C/C++, SystemC, VHDL and Verilog. There are also specialized tools. Heracles, for example, is an open-source, functional, parametrized, synthesizable multicore system toolkit [2]. It comprises several tools that support exploration of different design choices as well as hardware synthesis and program compilation. In particular, it supports fast exploration of multicore processors of different topologies, routing schemes, processing elements (cores), and memory system organizations.

Whereas in the aforementioned approach simulation is used within the optimization loop, it can also be used as a preprocessing step to gather data for later multi-objective optimization. Authors of [3], for example, assume a model written in SystemC TLM 2.0 and then perform a fully automatic optimization by simultaneous resource allocation, task binding, data mapping, and transaction routing for MPSoC platforms. Their hybrid optimization approach is based on an Evolutionary Algorithm and a pseudo boolean solver. A version of evolutionary algorithms, called the Strength Pareto Evolutionary Algorithm (SPEA) has been proposed in [4, 5]. It finds an approximation of the Pareto-optimal set for multi-objective optimization problems.

Constraint programming has been used for both optimization and multi-objective optimization. The time-energy trade-off has been studied for multi-layer memory architectures in [6]. The authors formalize the model using constraint programming and generate Pareto points for different memory configurations for each task. Then a specialized algorithm makes a hierarchical composition of the Pareto points for each task leading to possible execution scenarios. Constraint programming has also been used in [7] for generation of Pareto points for mapping of data and instructions for multimedia applications to scratch-pad memories. In this paper, we use a similar method for Pareto point generation, but focus on different metrics. A constraint-based DSE is proposed also in [8], for real-time applications implemented on MPSoCs. The authors formalize the mapping and scheduling problem together with other constraints on cost and performance. They minimize the throughput in their experiments and do not generate Pareto points.

Most research on DSE concentrates on system level decisions, such as mapping and scheduling, and papers on DSE for processor design usually concentrate on instruction selection, see for example [9]. There is not much work on processor design that tries to investigate different trade-offs between the number of functional units, execution pipelines and SIMD units. The paper [10] explores the problem of multiple SIMD units for GPUs but it is specific to graphic hardware. Our work addresses custom processors with SIMD and scalar units and tries to examine different configurations for obtaining a given throughput.

3 Background

We allocate this section to introduce existing techniques used in our approach.

3.1 Constraint programming

In this paper we extensively use constraint satisfaction methods implemented in the constraint programming environment JaCoP [11].

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 $\mathscr{D}_1 \times \mathscr{D}_2 \times \ldots \times \mathscr{D}_n$ 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.

A *global constraint* 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 [12]. In this paper we use intensively the global constraints named `cumulative` and `global_cardinality`.

The `cumulative` constraint [13] 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' start times, 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 `global_cardinality` constraint [14] is a generalization of the `all_different` constraint, which requires that every value is assigned to at most one variable. The constraint counts instead number of occurrences of given values in the variables list. The parameters are a list of values to be counted and two lists of FDVs. The first list is the *variable list* and the second list is the *counter* list. The counter list counts number of occurrences of values from the value list, in the list of variables.

A *solution to a CSP* is an assignment of a value to each variable from its respective domain, in such a way that all constraints are satisfied. The specific problem to be modeled will determine whether we need *just one solution*, *all solutions* or an *optimal solution* given some cost function defined in terms of the variables.

The solver is built using constraints' own consistency methods and systematic search procedures. *Consistency methods* try to remove inconsistent values from the domains in order to reach a set of pruned domains such that their combinations are valid solutions. Each time a value is removed from a FD, all the constraints that involve that variable are reevaluated. Most consistency techniques are not complete and the solver needs to explore the remaining domains for a solution using search.

Search usually assigns values from variable domains to the variables. It is implemented as depth-first-search. The consistency method is called as soon as the domains of the variables for a given constraint are pruned. If a partial solution violates any of the constraints, backtracking will take place, reducing the size of the remaining search space.

Standard search methods do not provide multi-objective optimization but can be used to build multi-objective search for Pareto points that is introduced in the next section and discussed more in details in section 5.1.

3.2 Pareto points

During the design process, different alternatives are explored. Design space exploration is therefore defined as a process of finding different solutions that offer trade-offs between various design parameters. It is usually achieved by carrying out a multi-objective optimization, meaning that it simultaneously optimizes more than one cost function. In such cases it makes little sense to talk about one single optimal solution, but instead we introduce the notion of *Pareto optimality*. In this research, we consider minimization of a multi-objective function and define *Pareto points* as a set of *non-dominated* solutions. A Pareto point is therefore a solution that has at least one lowest value for one cost function, i.e., it is not dominated by all other solutions. Pareto points form a trade-off curve (surface) called *Pareto curve (surface)*. Such a Pareto curve (surface) represents the available trade-offs between different optimization objectives.

The designer can then explore different trade-offs and select the Pareto solution that is most suitable for implementation with the current design constraints. Thus, a number of configurations that otherwise might be explored are discarded early on, narrowing the solution space, and helping the designer to identify potential solutions faster.

3.3 Modulo scheduling

Modulo scheduling is a common technique for increasing the throughput of a kernel. It involves finding a schedule that initiates iterations as soon as possible, taking into account dependencies and resource constraints, while also repeating regularly with a given interval (*initiation interval II* [15]). A more detailed expla-

nation of this technique including its performance on scheduling DSP kernels on a SIMD unit can be found in [1].

4 Problem definition

In this section we present the problem in its context, together with the assumptions surrounding the architecture abstraction and the target applications.

Our goal is to provide the processor designer with optimal alternative processor settings that meet throughput requirements for a given set of applications. An alternative is deemed *optimal* if it provides the required performance while keeping the resources included in the processor at the bare minimum.

The fact that there are alternatives is because there are several different kind of processing elements with different properties and capabilities. We assume that the processor can include several SIMD units and/or scalar units.

As the name suggests, each SIMD unit can run one type of operation over multiple data at a given time point, which is defined by the SIMD width. SIMD units to be included in the alternative solution have the same, but yet unknown width, which is also part of the exploration process. We assume that each unit can issue an instruction each clock cycle and is implemented as a hardware pipeline, incurring a latency depending on its length and complexity. In the example presented later in this paper we assume this latency to be seven clock cycles, based on our previous experience with a similar architecture [16]. For such an architecture the SIMD pipeline consists of one pre-processing, two core processing and two post-processing stages, along with one stage for load and one for store.

We assume that the scalar units can also issue one instruction each cycle, similar to the SIMD units. However, they do not have any pre- or post-processing capabilities, therefore have a lower latency (four clock cycles). Nevertheless, we assume that both the scalar units and the core processing stages for the SIMD units can carry out the same type of operations, covering all the operations present in the application set. This also means that in our current model, the pre- and post-processing stages are not employed, since the grouping of operations during scheduling (SIMDization) will only target core operations. However, extending the model to handle and generate pre- and post-processing specific operations (shuffling, masking, sorting) is feasible and part of our future work.

In this work we do not include memory or register considerations. We assume that enough multi-ported memory banks exist to cover the bandwidth demand to the processor and they are sufficiently large. When it comes do data alignment, specific data assignment constraints similar to those we presented in [16] can be added to the model. Furthermore, the size of the register file can be explored by evaluating the register pressure, as we show in [1]. Although all these parameters could be added to the Pareto space, as additional dimensions, for the sake of clarity,

the case study we present further employs only three parameters: the number of scalar units, the number of SIMD units, and the width of a SIMD unit.

As target applications, we consider kernels from digital signal processing (DSP) domain. Such kernels can be implemented differently depending of the computational model. For example, in an imperative language they are usually implemented as an iterative execution of a code sequence (a loop or a nested loop construct). In the dataflow model of computation [17], this is equivalent to the repetitive execution of an actor processing a data stream.

We confine ourselves to kernels that can be modeled as a *directed acyclic graph* (DAG) whose nodes represent basic operations and edges dependencies between them. Similar basic operations, which are also independent, can thus be grouped together and issued as one SIMD instruction.

5 Approach

In this section we present our approach to solving the problem defined in the previous section. The design space exploration flow we employ is depicted in Figure 1. As inpu,t we assume a set of applications the architecture is designed for. In multimedia and other streaming domains, the most computationally demanding parts are usually repetitive behaviours, or *kernels*, which are required to achieve a certain throughput. We assume that the throughput requirements and these kernels are extracted and are available for our use. Note that the throughput is dictated by the application, which often needs to provide a certain quality of service, e.g. samples per second. However, our kernel scheduling is carried out at clock cycle level, which is architecture dependent. Therefore, the designer has to also provide an estimate for the clock cycle, based on the used technology and own experience with hardware design.

Once these inputs are available, the exploration can start. For each kernel in the target application set, we generate and solve a constraint model to generate Pareto points (see Section 5.1) that meet the respective throughput requirements. Once the Pareto points for each such kernel are generated, they are delivered to the designer, who then merges these points into one architecture candidate, meeting the throughput requirements for the entire application set. The resulting architecture candidate depends on the design requirements (e.g. chip size, chip cost, power consumption) and the experience of the designer. Once the candidate is selected, the designer can implement a detailed model and evaluate it to ensure that the requirements are met. If the designer is not satisfied or prefers to experiment more with other candidates, another merge of the Pareto points provides a new candidate. Furthermore, the inputs can be refined based on this evaluation and the whole process re-run, in order to get more accurate results.

In this work we automate part of this flow as shown in the grey box from Figure 1. Note that this is a computationally intensive part, which would otherwise

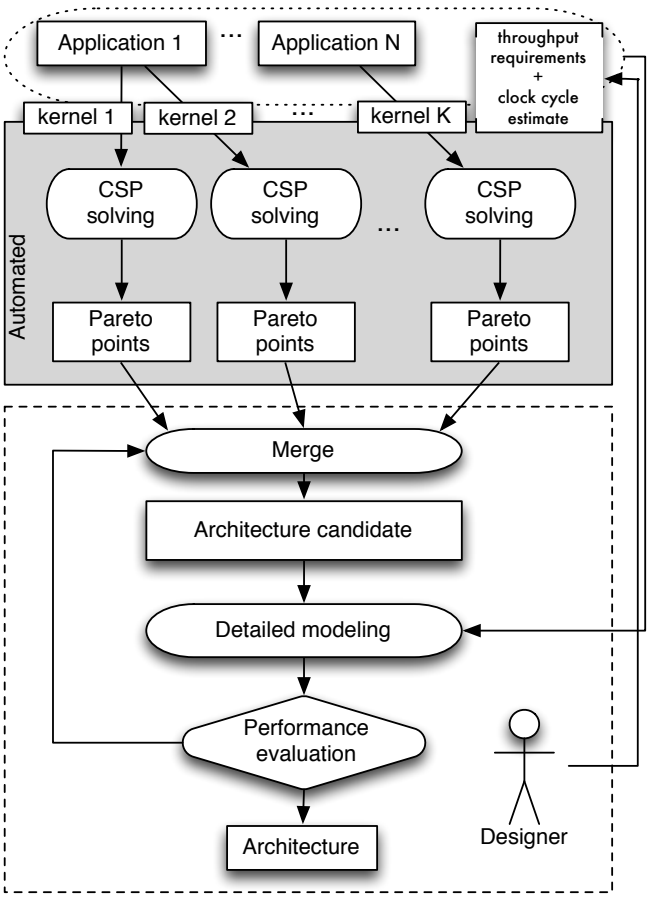


Figure 1: Design space exploration flow

require a long time to be carried out manually by a designer. Furthermore, generating the Pareto points for each kernel are tasks that can be carried out in parallel, leading to even faster design space exploration.

In the rest of this section we first describe how pareto points are generated with constraint programming and continue with modeling details.

5.1 Pareto points generation

To define multi-objective optimization more formally we assume that there exist several optimization criteria or cost functions denoted by $f_i(x), i = 1,\ldots,n$. We define a Pareto optimal solution as vector x, if all other vectors y have a higher value for at least one of the cost functions, i.e., $\forall y\ \exists i\ :\ f_i(y) > f_i(x)$.

The generation of Pareto points can be formalized using constraint programming. The idea is to start search with most relaxed constraints. When a solution is found, additional constraints are added to the solver. These constraints cut the part of the search space that can only have dominated points. Consider a two dimensional space with two criteria, cost and execution time. Assume that the solver finds a solution with cost c_i and execution time et_i. The newly added constraint should forbid solutions with greater cost or greater execution time and therefore constraint $cost < c_i \vee time < et_i$ is imposed. This constraint excludes any dominated point in the rest of the search but does not automatically eliminate dominated points previously found. Therefore it is necessary to explicitly remove such dominated points from the list of Pareto candidates.

The whole procedure for finding Pareto points is depicted in Listing 1. It assumes that we have a multi-objective optimization of n cost functions, denoted as $cost_i, 0 \leq i \leq n$. The procedure iteratively executes a depth first search method, called `DFS` in the listing. After each execution, the discovered n-dimensional point $(val_0,\ldots,val_n)$ is added to the set of Pareto points and a constraint that cuts dominated points is imposed. Dominated points are also removed from the set of already generated Pareto points, if such exist. The method has been used previously in [7] and recently formalized as a Pareto constraint [18].

```
vector pareto(𝒱, [cost_0, ..., cost_n])
paretoPoints ← ∅
solution ← DFS(𝒱)
while solution ≠ ∅
   (val_0, ..., val_n) ← solution
   paretoPoints ← paretoPoints ∪ (val_0, ..., val_n)
   impose cost_0 < val_0 ∨ ··· ∨ cost_n < val_n
   remove points dominated by (val_0, ..., val_n) from paretoPoints
   solution ← DFS(𝒱)
return paretoPoints
```

Listing 1: Pareto points generation algorithm

5.2 Modeling details

In the following, we detail the constraint model we built to generate the Pareto points for each application. Each point is in three-dimensional space with the number of available *SIMD Units*, the *SIMD Width*, and the number of *Scalar Units*, as the axes. *SIMD Units* can be up to N while *Scalar Units* can be up to M. *SIMD Width* on the other hand is dimensioned as powers of 2, up to 2^K.

As mentioned in Section 4, the applications are represented as DAGs and each node in the graph, representing an operation, is associated with several finite domain variables (FDVs):

- $start_i$ denotes when the node is scheduled to run.
- $modStart_i$ denotes the node's start time in the modulo schedule, formally $modStart_i = start_i \mod II$ where II is the initiation interval for the modulo schedule (see Section 3.3).
- $onSIMD_i$ is a boolean denoting whether or not the node is scheduled on a SIMD unit.
- $latency_i$ denotes the number of clock cycles to elapse before the output of node i is ready once it runs. This depends on $onSIMD_i$ as SIMD units and scalar units have different latencies.

The II is represented also as a variable. Since at the end of each *steady state* one iteration gets completed and since the length of the steady state is equal to II, the throughput can be represented as $1/II$. Therefore, to enforce the throughput requirement, we include $1/II \geq throughput$.

Each edge in the application DAG represents a data dependence between the connected nodes. Therefore, for each edge, we include an inequality constraint ensuring the precedence between the source and the destination nodes, so that the destination node will be scheduled after the source node produces its output. This is denoted by constraint (1)

$$\forall (i,j) \in E : start_i + latency_i \leq start_j \tag{1}$$

For the resource related constraints, we use the *modStart* variables. Once the modulo times are consistent, the *start* will comply automatically.

Each SIMD unit can only run one type of operation in one clock cycle. To ensure this, we first group operations based on their types, then count how many SIMD units a particular group needs each clock cycle, based on the count of nodes of that type scheduled to run on that clock cycle. Note that this count also is a variable depending on *onSIMD*, since each node can either run on a SIMD or scalar unit. Instead of using a counting constraint per clock cycle we use the

`global_cardinality` constraint. By grouping operations based on their types, we guarantee that only one type of operation is run on the same SIMD unit per clock cycle. Finally, we merge these groups in a `cumulative` constraint to limit the total number of SIMD units used in each clock cycle to the number of available SIMD units, which is one of the Pareto criteria.

The fact that we are not modeling each individual SIMD unit, but as a group of resources, lets us avoid symmetrical solutions leading to a growth in the search space exponential to the number of SIMD units available. It also simplifies the model by means of fewer variables.

Similar to the SIMD units, for all the scalar units, we include another `cumulative` constraint to limit the total number of scalar units used in each clock cycle to the number of available scalar units.

6 Case study

To demonstrate how our method can direct design space exploration, we provide a case study in which we provide a set of Pareto points for several DSP kernels, merge them into an architectural candidate and evaluate its performance to see that the throughput requirements are met.

6.1 Automated Pareto points generation

An architecture is defined by the following Pareto criteria:

- *SIMD Units* denotes the number of SIMD units available in the architecture.
- *Scalar Units* denotes the number of scalar units available in the architecture.
- *SIMD Width* denotes the width of each SIMD unit (i.e. number of parallel operations each unit can run).

The upper bounds for *SIMD Units*, *Scalar Units*, *SIMD Width* denoted as $(N,\ M,\ K)$ are set to $(4,\ 8,\ 3)$, respectively. Besides these characteristics, each SIMD unit is assumed to have a fixed number of pipeline stages that we assume for this case study to be seven and therefore have a latency of 7 clock cycles, while scalar units have a latency of 4 clock cycles.

Table 1 lists the details of the graph representations as well as the assumed throughput requirements and latency limits of the set of DSP kernels that we propose an architecture for in this case study. Explicitly, the application set includes Loeffler's inverse discrete cosine transformation (denoted only as IDCT in the rest) [19], an in house implementation of Modified Gram-Schmidt (MGS) based minimum mean squared error (MMSE) QRD algorithm [20] which is used in MIMO systems, and three other kernels from the Mediabench suite [21], namely elliptic

Application	**#nodes**	**#edges**	**Min thr.** (sample/cc)	**Max. lat.** (cc)
IDCT	48	63	0.050	100
QRD	106	157	0.020	400
EWF	34	47	0.100	100
JPEG_FDCT	134	169	0.021	200
MPEG_IDCT	114	164	0.014	200

Table 1: DSP kernels used in the case study

wave filter (EWF), forward discrete cosine transformation from JPEG and MPEG IDCT (not to be confused with Loeffler's IDCT).

Throughput and latency requirements are imposed by the application performance requirements. For different applications these parameters can be imposed by the architecture designer.

The Pareto optimization process is run on a MacBook Pro with 2.2 GHz Intel Core i7 processor and 8 GB RAM. The constraint model is implemented in the Minizinc 2.0 [22] constraint modeling environment and search is implemented in JaCoP [11].

Our CP implementation of the Pareto optimization may not always finish execution in a limited time. In such cases, the search space is not entirely examined and this fact has two consequences on generated points. First, there may be additional Pareto points that our search does not find. Second, the points found may not be Pareto optimal, i.e. they may be dominated by the points that are not yet found. In our case study only JPEG FDCT timed out in 10 minutes. Optimization for other kernels finished in less than 6 seconds, and therefore gave optimal Pareto points.

Fig. 2-5 depict the resulting Pareto points for all the kernels in Table 1. In these three dimensional figures axes (X, Y, Z) correspond to (*SIMD Units*, *SIMD Width*, *Scalar Units*) resprectively.

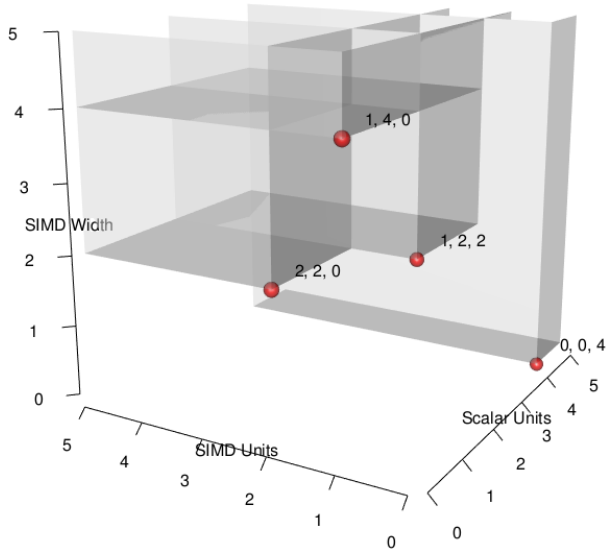


Figure 2: The Pareto points obtained for EWF. Exactly the same points were obtained for QRD.

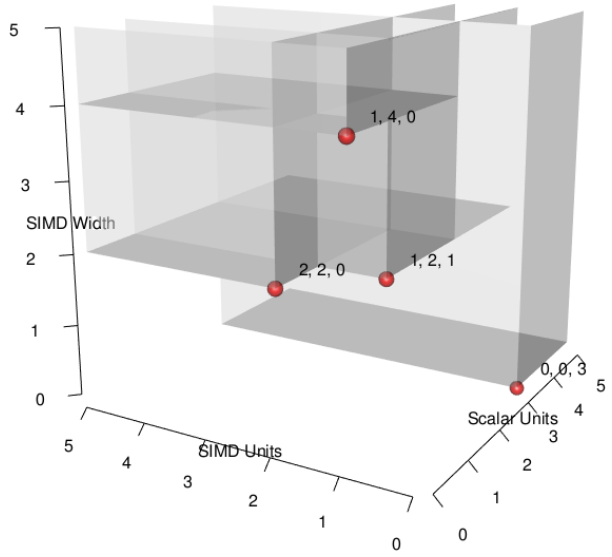


Figure 3: The Pareto points obtained for IDCT.

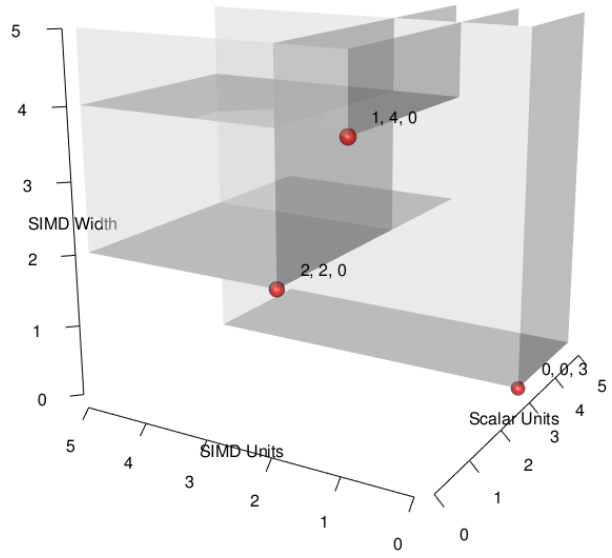


Figure 4: The Pareto points obtained for JPEG FDCT.

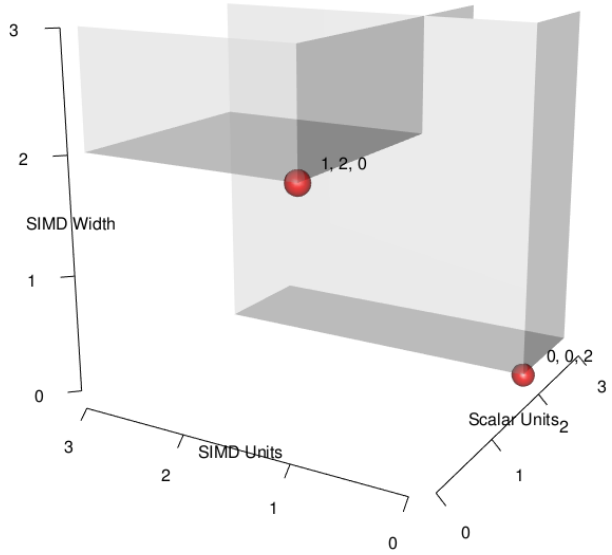


Figure 5: The Pareto points obtained for MPEG IDCT

6.2 Candidate selection and evaluation

As mentioned earlier, after the generation of the Pareto points, the designer needs to merge them and come up with a candidate architecture with the Pareto criteria that meets every kernel in the target set. Consequently, the designer has to make sure that the proposed candidate meets the application requirements. In this section, we assume the designer's role and demonstrate how the designer-dependent part of our design flow can work.

Inspecting the Pareto points, we can see that $(1,\ 4,\ 0)$ is a common point for the first four kernels while it is dominated by $(1,\ 2,\ 0)$ for MPEG IDCT, which means that in principle it should meet the performance requirements for all five kernels. Note that $(2,\ 2,\ 0)$ is also a suitable option but we prefer $(1,\ 4,\ 0)$ over $(2,\ 2,\ 0)$ since we assume two tight SIMD units are costlier than one larger SIMD with the same processing power.

In order to evaluate this choice, we scheduled each kernel and measured the requirements together with the utilization, as shown in Table 2. We let the solver run one minute to maximize the throughput. A comparison of Tab. 1 and 2 shows that all kernels meet their requirements on the proposed architecture. Besides, with the maximization of throughput in scheduling, the SIMD unit is utilized over 85%, except for MPEG IDCT. This deviation can be explained by the fact that this kernel needs only $(1,\ 2,\ 0)$ as Pareto criteria, as shown in Figure 5, and the rest of the extra width in the proposed SIMD unit is therefore underutilized.

Since the schedules result in a margin of improvement w.r.t. throughput, the designer can use this leeway to decrease the clock frequency if necessary (e.g. to decrease power consumption), without invalidating the requirements.

Application	**Throughput** (sample/cc)	**SIMD Util.** %	**Latency** (cc)
IDCT	0.071	86	42
QRD	0.037	98	287
EWF	0.111	94	98
JPEG_FDCT	0.026	87	91
MPEG_IDCT	0.022	62	112

Table 2: Evaluation of the candidate architecture (1,4,0)

7 Conclusions

We presented a method for improving the design space exploration targeting custom processor architectures, by automating the extraction of solution candidates. Our method, starts from a set of applications with computationally intensive kernels and throughput requirements. The problem is modelled in a constraint pro-

gramming environment and solved efficiently as a set of Pareto optimization subproblems. As a result, it proposes a number of possible processor configurations for each application, such that the requirements are met and resources minimized. As resources we mainly focus at this point on the number of SIMD units, their width and the number of scalar units. It is then up to the designer to select the best combination of these solutions and go further to more detailed modeling and evaluation of these design candidates.

We illustrate our approach using a set of five kernels from the multimedia domain. The analytical evaluation of the most promising solution candidate suggested by our design flow shows that it improves over the performance requirements with a considerable margin and provides very high utilization. Whereas a corresponding manual attempt can take months of labor by experienced designers, our method can suggest and analytically evaluate candidates in minutes, efficiently extracting a set of feasible solutions with minimal resources.

In the future we plan to experiment with adding more design parameters, as well as modeling the architecture more accurately, including storage and processing units constraints.

References

[1] Mehmet Ali Arslan, Flavius Gruian, and Krzysztof Kuchcinski. "A comparative study of scheduling techniques for multimedia applications on SIMD pipelines". In: *Proceedings of the DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS 2015)*. 2015. URL: http://arxiv.org/pdf/1502.07447.pdf.

[2] Michel A. Kinsy, Michael Pellauer, and Srinivas Devadas. "Heracles: A Tool for Fast RTL-based Design Space Exploration of Multicore Processors". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA: ACM, 2013, pp. 125–134. URL: http://doi.acm.org/10.1145/2435264.2435287.

[3] Martin Lukasiewycz, Martin Streubuhr, Michael Glass, Christian Haubelt, and Jürgen Teich. "Combined System Synthesis and Communication Architecture Exploration for MPSoCs". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009, pp. 472–477. URL: http://dl.acm.org/citation.cfm?id=1874620.1874737.

[4] Eckart Zitzler and Lothar Thiele. "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach". In: *Trans. Evol. Comp* 3.4 (Nov. 1999), pp. 257–271. URL: http://dx.doi.org/10.1109/4235.797969.

[5] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. "SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization". In: *Evolutionary Methods for Design, Optimisation, and Control*. CIMNE, Barcelona, Spain, 2002, pp. 95–100.

[6] Radoslaw Szymanek, Francky Catthoor, and Krzysztof Kuchcinski. "Time-Energy Design Space Exploration for Multi-Layer Memory Architectures". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 10318–. URL: http://dl.acm.org/citation.cfm?id=968878.969089.

[7] Ali R. Iranpour and Krzysztof Kuchcinski. "Design space exploration for optimal memory mapping of data and instructions in multimedia applications to Scratch-Pad Memories". In: *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*. Oct. 2009, pp. 89–95.

[8] Kathrin Rosvall and Ingo Sander. "A Constraint-based Design Space Exploration Framework for Real-time Applications on MPSoCs". In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '14. Dresden, Germany: European Design and Automation Association, 2014, 326:1–326:6. URL: http://dl.acm.org/citation.cfm?id=2616606.2617072.

[9] Unmesh D. Bordoloi, Huynh Phung Huynh, Tulika Mitra, and Samarjit Chakraborty. "Design space exploration of instruction set customizable MPSoCs for multimedia applications". In: *Embedded Computer Systems (SAMOS), 2010 International Conference on*. July 2010, pp. 170–177.

[10] Dana Schaa and David Kaeli. "Exploring the multiple-GPU design space". In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. May 2009, pp. 1–12.

[11] Krzysztof Kuchcinski. "Constraints-Driven Scheduling and Resource Assignment". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 8.3 (July 2003), pp. 355–383.

[12] Willem-Jan van Hoeve and Irit Katriel. "Handbook of Constraint Programming". In: Foundations of Artificial Intelligence. Elsevier Science, 2006. Chap. Global Constraints.

[13] Abderrahmane Aggoun and Nicolas Beldiceanu. "Extending chip in order to solve complex scheduling and placement problems". In: *Mathematical and Computer Modelling* 17.7 (1993), pp. 57–73.

[14] Jean-Charles Regin. "Generalized Arc Consistency for Global Cardinality Constraint". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*. AAAI'96. Portland, Oregon: AAAI Press, 1996, pp. 209–215. URL: http://dl.acm.org/citation.cfm?id=1892875.1892906.

[15] B. Ramakrishna Rau. "Iterative modulo scheduling". In: *International Journal of Parallel Programming* 24.1 (1996), pp. 3–64.

[16] Mehmet Ali Arslan, Krzysztof Kuchcinski, Flavius Gruian, and Yangxurui Liu. "Programming Support for Reconfigurable Custom Vector Architectures". In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '15. San Francisco, California: ACM, 2015, pp. 49–57.

[17] Edward A. Lee and Thomas M. Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (May 1995), pp. 773–801.

[18] Renaud Hartert and Pierre Schaus. "A Support-Based Algorithm for the Bi-Objective Pareto Constraint". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* 2014, pp. 2674–2679.

[19] Christoph Loeffler, Adriaan Ligtenberg, and George S. Moschytz. "Practical fast 1-D DCT algorithms with 11 multiplications". In: *ICASSP-89, International Conference on Acoustics, Speech, and Signal Processing,* May 1989, 988–991 vol.2.

[20] Peter Luethi, Andreas Burg, Simon Haene, David Perels, Norbert Felber, and Wolfgang 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.

[21] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems". In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society. 1997, pp. 330–335.

[22] The G12 Team. *Specification of MiniZinc*. Tech. rep. NICTA, Victoria Research Lab, Australia, November 2014.

CODE GENERATION FOR A SIMD ARCHITECTURE WITH CUSTOM MEMORY ORGANISATION

Abstract

Today's multimedia and DSP applications impose requirements on performance and power consumption that only custom processor architectures with SIMD capabilities can satisfy. However, the specific features of such architectures, including vector operations and high-bandwidth complex memory organization, make them notoriously complicated and time consuming to program. In this paper we present an automated code generation approach that dramatically reduces the effort of programming such architectures, by carrying out instruction scheduling and memory allocation based on a constraint programming formulation. Furthermore, the quality of the generated code is close to that of hand-written code by an experienced programmer with knowledge of the architecture. We demonstrate the viability of our approach on an existing custom heterogeneous DSP architecture, by compiling and running a number of typical DSP kernels, and comparing the results to hand-optimized code.

Mehmet Ali Arslan, Krzysztof Kuchcinski and Flavius Gruian - Dept. of Computer Science, Lund University
Andréas Karlsson - Dept. of Electrical Engineering, Computer Engineering. Linköping University
*Based on the paper that will be published on the proceedings of the Conference on Design & Architectures for Signal & Image Processing October 12-14, 2016, Rennes, France

1 Introduction

Embedded systems are a special class of computing systems with very specific requirements on performance, cost and power/energy consumption, which are often addressed via special design and architectural choices. In particular, custom processors are often employed, as they bring ways to leverage and finely tune the application parallelism in order to meet the performance at acceptable power budgets. For DSP and multimedia applications, which are the type of applications we target in this paper, data parallelism is usually supported by SIMD instructions and specialized high-bandwidth memory architectures.

However, the performance introduced by such architectural solutions comes with a trade-off in programmability. Traditional compilers are not built to exploit the irregular structure and specific features available in such architectures. Therefore, using standard techniques and tools to compile from a high level language comes at the price of very low quality of the generated code.

The preferred alternative is to write machine code by hand. However, there are several problems with this approach. First of all, coding becomes extremely time consuming and tedious. The programmer has to select the instructions to implement the program. For architectures with SIMD-like features, this means bundling smaller similar operations into vector instructions. To utilize the processor efficiently and increase application throughput, the programmer also needs to come up with a schedule that parallelizes the code as much as possible, while respecting the resource and memory constraints. It can take many man-hours to write 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 information is available only to the architect of the processor. Even for the architect, the overwhelming amount of information to be taken into account while writing machine code by hand, makes this a tedious and error-prone process.

Our goal in this paper is to increase the programmability of such custom architectures without losing performance compared to the hand-written code (by the architect), by automating the code generation process. As our target platform for this study, we use an existing custom architecture with vector processing capabilities, built especially for running DSP algorithms efficiently [1]. The architecture offers SIMD instructions that can execute up to four operations in parallel and a specialized memory architecture that can feed enough data to the SIMD unit.

In our work, we experiment with computationally intensive parts of programs that are run many times for each piece of data (i.e. kernels). Shortening the schedule for these parts can drastically increase the overall application performance, and therefore, aggressive optimization techniques are beneficial even if they require longer compilation times. In this study, we consider such kind of programs, represented as feed forward dataflow graphs.

2 Related Work

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

Instruction selection and scheduling for a given processor or multi-processor are complex problems known to be NP-complete. Special attention has been given recently to custom architectures that have non-regular instruction sets as well as non-regular memory organizations. This makes it difficult to use well-known compiler infrastructures, such as LLVM [2]. An extensive survey about instruction selection by Blindell [3] is an invaluable text for further reading on the subject.

Different aspects of efficient code generation for SIMD architectures have been studied. Wu et. al. explore memory alignment issues [4] and propose a method to simdize loops with runtime alignment and conversions between data of different sizes. Authors of [5] recognize the problem of code generation for SIMD architectures for irregular kernels. They study disjoint memory references, arbitrarily misaligned memory references, and dependence cycles in loops and propose a heuristic method to generate SIMD code. They extract both inter- and intra-iteration parallelism, taking data reorganization overhead into consideration, and place data with help of data reorganization code. Our approach works as well for irregular loops, making it possible to generate efficient code regardless if the original code has regular or irregular references. We make use of the custom nature of the architecture and try to minimize data permutations while not employing data reorganization code.

The polyhedral model formalizes data dependencies in loops, in concise mathematical notations and is used to define different code transformations that facilitate code generation, not only but also for SIMD architectures. Authors of [6] use, for example, the polyhedral model for code generation for SIMD architectures. They propose a 3-step framework that facilitates SIMD execution of programs. Adaptive loop tiling approach based on polyhedral transformations is also introduced in [7]. We do not employ loop transformations based on polyhedral model but they can be provided for our framework as pre-processing.

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

Bednarski [8] 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 optimum, 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 constraint programming (CP) approach, developed during recent years for different purposes, is one of the methods used for solving these problems optimally. Beek and Wilken [9] use CP to optimally schedule basic block instructions on single-issue RISC processors. They address arbitrary latencies for instructions.

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. Our approach addresses code generation for vector processors and differs mainly in the focus on data memory allocation and data accesses for these architectures, 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.

The authors of [12] developed an approach to register allocation using constraint programming. Their unified optimization framework simultaneously considers the impact of loop unrolling and instruction scheduling. This is achieved by an instruction tiling approach where instructions within a loop are represented along one dimension and innermost loop iterations along the other dimension. The approach makes it possible to exploit regularity along the loop dimension. Our approach does not depend on regular loops and can handle arbitrary data references.

Another optimal method for instruction scheduling and register allocation is presented in [13], 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 [14], which is a well established software pipelining technique.

Our previous work [15, 16] uses CP for instruction selection and scheduling for reconfigurable processor extensions. We address both complex instructions and SIMD reconfigurable architectures. Instruction selection for complex instructions employs a custom global constraint for sub-graph isomorphism while SIMD code generation assumes specifications in domain specific language (DSL). In this work we focus less on instruction selection and more on scheduling with memory allocation. We do not assume regular loops either and can handle irregular data accesses. Our input is a graph based internal representation that can be generated from most high-level languages.

3 Background

3.1 Constraint programming

In this paper we extensively use constraint satisfaction methods implemented in the constraint programming environment JaCoP [17].

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 $\mathscr{D}_1 \times \mathscr{D}_2 \times \ldots \times \mathscr{D}_n$ 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.

A *global constraint* 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 [18]. In this paper we use, among others, intensively the global constraints named `cumulative`, `diff2` and `regular`.

The `cumulative` constraint [19] was originally introduced to specify 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. The `diff2` [20] constraint is designed to model the placement of rectangles in two dimensional space in such a way that they do not overlap.

The `regular` constraint [21] is defined for a sequence of variables and a deterministic finite automaton (DFA). It requires that the corresponding sequence of values taken by these variables belong to a given regular language, defined by the DFA.

A *solution to a CSP* is an assignment of a value to each variable from its respective domain, in such a way that all constraints are satisfied. *Consistency methods* try to remove inconsistent values from the domains in order to reach a set of pruned domains whose combinations are valid solutions. Most consistency techniques are not complete and the solver needs to explore the remaining domains for a solution using search.

Search usually assigns values from variable domains to the variables using depth-first-search. The consistency method is called as soon as the domains of the variables for a given constraint are pruned. If a partial solution violates any of the constraints, backtracking will take place, reducing the size of the remaining search space.

3.2 Target architecture: ePUMA

ePUMA is a heterogeneous digital signal processing (DSP) architecture [1], which includes a master processor that is responsible for overall application execution, and which may off-load DSP computing tasks to multiple compute clusters. The master processor and compute clusters communicate with off-chip main memory and with each other through a star network and a bi-directional ring network. A mailbox system can be used for notification and synchronization. Fig. 1 shows an overview of the architecture, with the components of interest for this work highlighted.

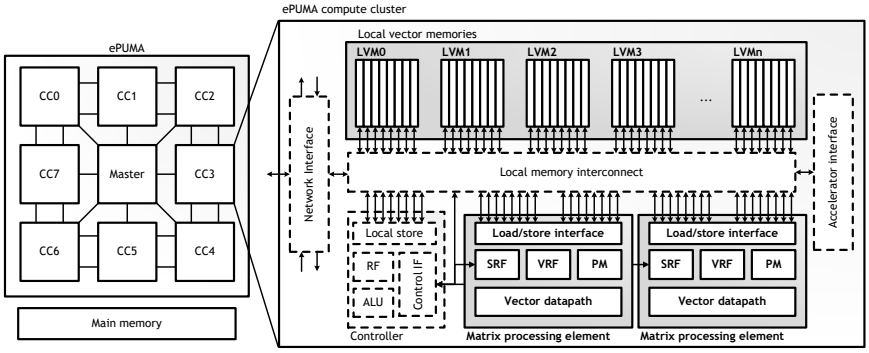


Figure 1: Overview of ePUMA architecture.

A computing cluster contains a local controller, multiple vector DSP compute processors called Matrix Processing Elements (MPEs) and a set of local vector memories (LVMs). Each MPE can be assigned two of the local memories at a time for processing. The memories may be reassigned to exchange data between cores. An MPE contains a complex datapath structure, with 16 multipliers and three levels of arithmetic units, to accelerate general and application-specific computing patterns. The processor may operate on vectors of arbitrary length directly in the local memories, processing 128-bits per clock cycle and vector operand. The focus of this work is limited to code generation for a single MPE.

Each local memory is a single-ported multi-bank memory, which allows conflict and penalty-free parallel memory access even in presence of unaligned and out-of-order memory access patterns, as long as any vector memory access only requests from each memory bank once every clock cycle. Since the memory access patterns of most DSP algorithms are predictable, data can often be prepared to guarantee conflict-free access, which eliminates the data access bottleneck present in many other SIMD and vector architectures. Generating conflict-free memory accesses is one of the main challenges addressed in this work.

4 Problem definition

To achieve our goal, we employ the following process, depicted in Figure 2, which makes use extensively of constraint programming (CP), proven to be effective for solving scheduling and allocation problems [17]. We assume that programs are compiled to an intermediate representation (IR) which is used together with architecture constraints to generate a constraint model that defines the problem. This model combines operation scheduling with memory allocation in a CP model, since these stages are highly dependent on each other. The architecture constraints include parameters, such as width of SIMD unit, number of memories and registers, latencies for operations as well as specific characteristics of memory/register access patterns and access latencies. As output from the constraint solver, we obtain an instruction schedule along with memory allocation that contains all the information needed to generate the machine code. To analyse the performance of the obtained code, we employ a simulator for the target architecture to run the code.

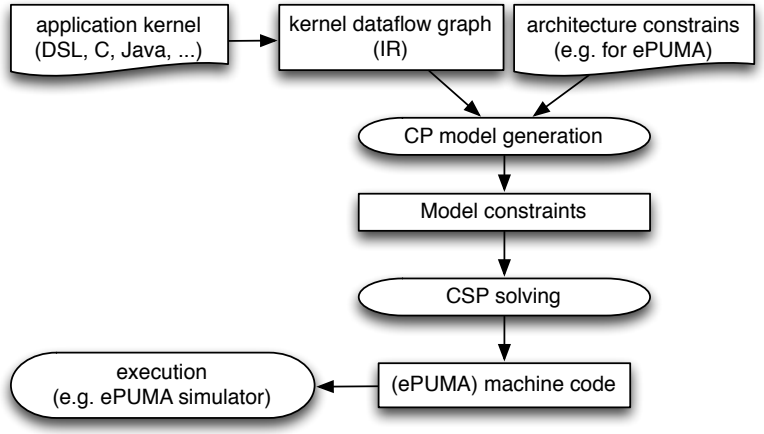


Figure 2: Code generation flow for SIMD unit.

4.1 Architectural assumptions

The targeted version of the architecture implements binary operations. Therefore we assume all operations to have the form:

```
op  dst src1 src2
```

where,`op` is the operation code and `dst` represents the destination of the outputs, while `src1` and `src2` represents the first and second operands of the operation

respectively. As we have a SIMD-processor, there can be up to SW operands in one instruction. Figure 3 depicts an example instruction for the targeted version of the architecture, where $SW = 4$. The SIMD logic works point-wise and aligns operations to *lanes* 0 to $SW - 1$. Note also that we assume fixed order of operands, defined by the input program. The operands in `dst`, `src1` and `src2` can reside

$$\begin{array}{l} p = a + e \\ q = b + f \\ r = c + g \\ s = d + h \end{array} \quad \Rightarrow \quad \texttt{sum[p,q,r,s][a,b,c,d][e,f,g,h]}$$

Figure 3: SIMD logic

either in the memory or a register, but each operand group has to come from the same memory/register. Registers provide faster access while memory provides more space and flexibility. There are two 4-bank memories and 8 registers available.

Each multi-bank memory allows reading a full vector in one clock cycle provided that the access does not involve reading multiple elements from the same bank (which constitutes a *bank conflict*). Some access patterns (henceforth called *regular access*) are supported implicitly through the hardware implementation while other patterns can be used through the help of a *permutation vector* stored in the program memory as long as it does not entail any bank conflicts.

The architecture can issue an instruction each clock cycle but instruction latency depends on several factors. Different operation types can have different latencies. For example, we assume that the default latency for a multiplication is 4 clock cycles, i.e. the output of a multiplication is ready to use 4 cycles after its issue. While the same figure for an addition is 1 clock cycle. This default latency is extended when one of the following occurs:

- *Writing back to memory:* Results in one clock cycle additional latency.
- *Bank conflict:* Results in one clock cycle additional latency per conflict, per bank.
- *Memory conflict:* Both *src*1 and *src*2 are read from the same memory. As the memory is single-ported, this adds one clock cycle latency.
- *Too many permutations:* Each irregular access needs a permutation vector to be defined and kept in the program memory. The architecture provides a way to avoid the permutation penalty if there is only one read permutation in the pipeline. But when there is more, this adds to the latency.

Registers on the other hand cause no such additional latency or delay penalties. However, they only allow regular access.

4.2 Application assumptions

The application to run on the architecture is written in a high-level language and translated into to an intermediate representation in form of 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.

In the rest of the section we will use the simple application depicted with its corresponding graph in Figure 4 as our running example to illustrate the key aspects of our model. This small application consists of 4 multiplications and 2 additions.

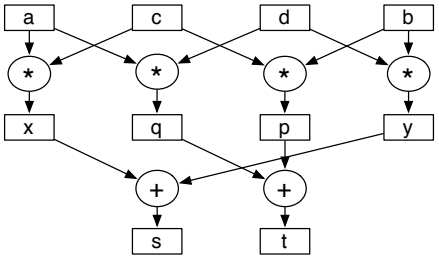


Figure 4: Running example

We assume that the data placement for the inputs to the application is fixed beforehand, and all input data reside in memories. In case of an operation on two sets of data (e.g. matrix multiplication) we assume that each set reside in one single memory, placed in consecutive addresses. For our running example, this means $[a, b]$ reside in addresses $[0, 1]$ in the first memory and $[c, d]$ reside in addresses $[0, 1]$ in the second memory.

5 Approach

5.1 Modeling details

As in our previous work [16, 15], we have precedence constraints that ensure the data dependencies are respected. Similarly, we use `cumulative` to group at most SIMD-width (SW) operations for running simultaneously. Another recurring constraint is `diff2`, which we use for memory and register allocation to allow reuse of these data locations while guaranteeing non-overlapping lifetimes.

The big difference in this work is the modeling of memory accesses. As mentioned earlier, the memory can be accessed regularly *without* any latency penalty,

t	dst				src1				src2			
	lane 0	lane 1	lane 2	lane 3	lane 0	lane 1	lane 2	lane 3	lane 0	lane 1	lane 2	lane 3
0	(r0,0)	(r0,1)	-1	-1	(m0,0)	(m0,1)	-1	-1	(m1,0)	(m1,0)	-1	-1
1	(r1,0)	(r1,1)	-1	-1	(m0,1)	(m0,0)	-1	-1	(m1,1)	(m1,1)	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
⋮												
5	(m1,0)	(m1,1)	-1	-1	(r0,0)	(r0,1)	-1	-1	(r1,0)	(r1,1)	-1	-1

(a) Memory schedule. Unused slots are assigned -1.

```
mul [x,p] [a,b] [c,c]
mul [y,q] [b,a] [d,d]
-
sum [s,t] [x,p] [y,q]
```

(b) Corresponding instruction schedule.

Figure 5: One of the possible optimal schedules for the example in Figure 4. Outputs are ready in t=7.

or more flexibly *with* possible latency penalty. Any *irregular* access needs also a permutation vector to define the memory access for the machine. Therefore we have to identify for each instruction, the regularity of access or lack thereof, in order to decide whether a permutation vector is necessary.

One method is to ignore and not model the permutation vectors and calculate them as a post-processing step, after the operation schedule memory/register assignment is decided. However this results in schedules that need many permutation vectors which in turn causes an abundance in latency penalties, degrading the performance of the generated code.

Instead, we choose to incorporate the permutation vectors in our constraint model, so that the solver is aware of the effects scheduling and data placement decisions have over permutation vectors, during the decision process. This way the solver can make decisions leading to less permutation vectors whenever possible, and avoid the additional latency.

In most of this section we will describe the model and constraints around permutation vectors.

5.1.1 Memory schedule

In order to reason about regularity, we need to know which operations are bundled together, so that we can analyze which locations are accessed simultaneously. However this is known only when the operation scheduling is decided, i.e. depending on when an operation is scheduled, the groupings change. Therefore we need a data structure of FDVs that keeps track of the simultaneous read and write accesses for each time step depending on the schedule, which will be updated by constraints during scheduling and memory/register assignment.

Figure 5b depicts the data structure we use for this purpose, filled with a possible schedule for our running example from Figure 4. This *memory schedule* consists of three matrices, namely dst, $src1$ and $src2$, with the same structure, indexed by *time* and *lane*. The elements of the matrices are connected to operation scheduling and memory/register assignment with constraint (1), where $opNodes$ denotes the set of all operation nodes, $start_o$ represents the issue time of operation o and $lane_o$ the lane it is assigned to on the SIMD core. By this constraint, each operation sets the slot indexed with $(start_o, lane_o)$ in each matrix to the addresses of its inputs and output.

$$\begin{aligned} &\forall o \in opNodes: \\ &\quad dst_{start_o,lane_o} = address_{out_o} \\ &\quad src1_{start_o,lane_o} = address_{in1_o} \\ &\quad src2_{start_o,lane_o} = address_{in2_o} \end{aligned} \tag{1}$$

lane variables are used for reasoning over the order among simultaneous operations, which is necessary for deciding over permutation vectors. $address_d$ on the other hand represents the location of data d in the *flattened* data space. This

includes the address spaces of 2 memories and 8 registers. This flat view is useful for expressing arithmetic constraints and constraints concerning data lifetimes.

Throughout the paper we use the tuple $(m_d, offset_d)$ to represent the memory/register assignment and offset assignment within that memory/register for the data node d respectively. Note that $(m_d, offset_d)$ is only a different view of $address_d$; they are not necessary but they make our model more concise. In Figure 5b we use symbolic names such as $m0$ or $r0$ for memory/register assignment instead of m_d to improve the readability of the figure. Values $\{0,1\}$ model memories and 2..9 model registers. Since register size is equal to vector size (also SW), $offset_d$ is limited to values 0..3. Note also that the accesses to the matrices in constraint (1) are done with FDVs as indices, therefore they are translated into element constraints in the model.

While constraint (1) sets the elements of the memory schedule based on the operations scheduled for each time and lane, constraint (2) sets the slots where no operations are scheduled, to -1. This is achieved by counting the number of -1s in each row and equating that to $SW - ops\_in\_t_t$, where $ops\_in\_t_t$ is the number of operations scheduled on t. $ops\_in\_t_t$ is maintained by a *global cardinality constraint*, which counts the occurrence of each t among operation start times and sets $ops\_in\_t_t$ to this count. Note that in constraint (2) and the rest of the paper, we use single-indexing as a shorthand for representing the vectors of memory/register accesses that happen in t as dst_t, $src1_t$ and $src2_t$.

$$\begin{aligned} &\forall t \in 0..MAX_T : \\ &\quad count(-1, dst_t) = SW - ops\_in\_t_t \\ &\quad count(-1, src1_t) = SW - ops\_in\_t_t \\ &\quad count(-1, src2_t) = SW - ops\_in\_t_t \end{aligned} \tag{2}$$

5.1.2 Access constraints

With the help of the memory schedule, we can now reason about accesses to memory and registers at each time step. The goal here is, for each access, to define whether it is regular (defined by (3)) or if we need a permutation vector. This way we can try eliminating permutation vectors to decrease the latency penalty that may occur because of them, and if not calculate the permutation vector necessary for code generation.

For this study we define an access as *regular* if it is increasingly or decreasingly consecutive, or constructing a vector through repetition of a single scalar or a consecutive half vector. Note that repetition is allowed only for $src1$ or $src2$, since having repetition in dst would mean writing to the same slot more than once, simultaneously. For a full vector operation for $SW = 4$, this would mean the alternatives listed in (3) for any $src1_t$ or $src2_t$, assuming that the access starts with address a. The architecture additionally allows fixed steps of arbitrary size as long as they do not result in bank conflicts but our model does not support this yet.

$$
\begin{aligned}
&[a, a, a, a] \\
&[a, (a+1), a, (a+1)] \\
&[a, (a+1), (a+2), (a+3)] \\
&[a, (a-1), (a-2), (a-3)] \\
&[a, (a-1), a, (a-1)]
\end{aligned} \tag{3}
$$

The regular access alternatives in (3) can be seen as comprising a *language* that can be represented by a concise *deterministic finite automaton* (DFA), if we transform each access vector based on its first access and introduce several additional constraints about where the -1 entries can appear in an access vector. The transformation is done by constraint (4) where $transform(row)$ transforms the row at hand. To simplify the reasoning around regularity and keep the regularity DFA concise, we push the -1 entries to the end of each row by constraint (5), where m_{row_i} is set to a memory/register assignment or -1 according to constraints (1) and (2). With this transformation we can represent the language of regularity with a DFA with 13 states and 29 edges (*dfa* in constraint (4)).

$$
\begin{aligned}
&\forall t \in 0..MAX_T,\ row \in \{dst_t, src1_t, src2_t\},\ i \in 0..SW-1 : \\
&transform(row)_i = \begin{cases} -1 & \text{for } row_i = -1 \\ |row_i - row_0| & \text{for } row_i \geq 0 \end{cases}
\end{aligned} \tag{4}
$$

$$
decreasing(m_{row_i}) \tag{5}
$$

For each access vector (i.e *row* in constraints (6, 7, 8)) we augment its transformed version with a boolean FDV named $p1_{row}$, representing its regularity. This augmented vector is fed to the DFA which sets $p1_{row}$ to 0 if the transformed vector is regular and to 1 otherwise (see constraint (6)). A side effect of transformation with absolute values is that we can end up with a regular transformed row when in fact the original row is not regular. For example, if the row is $[4,5,4,3]$, it's transformation will be $[0,1,0,1]$ which is regular according to (3). To alleviate this side effect, we introduce another boolean FDV $p2$ for each row, which is set to 0 when the first element is either the smallest element (except -1) or the largest, and to 1 otherwise. This is captured in constraint (7). A permutation vector is necessary for a row when any of $p1$ or $p2$ to is set to 1 (constraint (7)).

$$\forall t \in 0..MAX_T,\ row \in \{dst_t, src1_t, src2_t\}:$$
$$\texttt{regular}(transform(row) \oplus p1_{row},\ dfa) \tag{6}$$
$$p2_{row} = \ row_0 \neq \texttt{minEx}(row, -1) \wedge row_0 \neq \texttt{max}(row) \tag{7}$$
$$permVector_{row} = p1_{row} \vee p2_{row} \tag{8}$$
$$m_{row} \in REG \implies permVector_{row} = 0 \tag{9}$$

Permutation is only available for memory accesses, hence we force regularity for register accesses with constraint (9). We reinforce constraint (9) by letting the lane of an operation decide the offset of a register access:

$$\forall o \in opNodes, d \in o_{ins} \bigcup \{o_{out}\}:$$
$$lane_o = (offset_d \mod SW) \tag{10}$$

Together with constraint (5), this makes it impossible to access register elements randomly. Any access has to start from the first element.

5.1.3 Writeback and read conflict for memory accesses

Besides permutation vectors, accessing the memory too often in an instruction may incur latency penalties, as mentioned in Section 4.1. Figure 6 depicts another possible schedule for the example from Figure 4. There are several differences to note comparing this solution to the solution in Figure 5b. Instead of using any registers, this solution groups all 4 multiplications together by using a permutation vector in $src2$ and putting the results to $m1$. These results have to be summed with each other, therefore in time step 5 $m1$ is accessed twice for reading the operands. The writeback to $m1$ in $t = 0$ causes one clock cycle additional latency, hence the

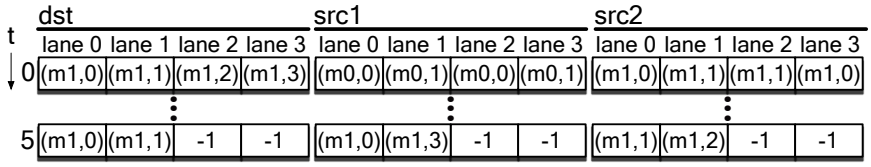


Figure 6: Memory schedule for the example in Figure 4 for a sub-optimal schedule. Outputs are ready in t=8

next instruction that is dependent on its results is issued on $t = 5$. The double read from $m1$ and again the writeback to $m1$ in $t = 5$ adds two clock cycles to the latency of the last instruction, making the total length of the schedule 8 clock cycles, versus the 7 clock cycles of the schedule in Figure 5b.

When modeling the latency for each operation, we add these latency penalties conditioned on the data placement of its inputs and output.

6 Experiments

In this section we present our experiments to evaluate the performance of our code generation method. The point is to demonstrate that the code generated by our method, which takes a program written in a high-level language and automates instruction scheduling and data placement with permutation vectors, performs comparably well against hand-written code by one of the architects of the target architecture.

6.1 Assumptions

We assume that the input data for each application are complex numbers with real and imaginary parts. This is why the $SW = 4$ while normally the architecture divides the width to 8 words of 16 bytes. We generate assembly code for the architecture and run it on its cycle-accurate simulator.

6.2 Applications

The main application we target is square *matrix multiplication*, used commonly in DSP applications such as MIMO [22]. We experimented with 3 different sizes for input matrices: 2x2, 3x3 and 4x4. 2x2 and 4x4 fits the SW perfectly while 3x3 is trickier to program for maximal utilization of the SIMD core and the registers with size equal to SW.

Other than matrix multiplication, we target two more kernels common for DSP applications, FIR-filter with varied sizes and quantization. FIR-filter can be seen as the repetition of an innermost loop. The purpose of modifying the size is to increase utilization and performance through exposing more parallelism, at the cost of increased compilation time and code size.

6.3 Results

We outline results in performance comparison in Tab. 1. The size of each kernel's graph is given in parenthesis in the leftmost column. The timeout for the constraint solver is set to 10 minutes. For the smaller examples, 2x2 matrix multiplication, 4x4-tap FIR-filter, the solver proves optimality in less than a minute. For the others the solution reported in the table is found within the first 2 minutes and rest of the time is spent for searching for a better alternative. Therefore it is fair to assume a compile time of 2 minutes overall.

For matrix multiplication, we match the hand-written code in performance except in the largest instance, namely 4x4. Even in 4x4 the overhead is reasonable, considering the fact that the hand-written code is by an architect, whose detailed knowledge and experience over the architecture gives him an edge.

For FIR-filter our method's performance degrades slightly, especially for 8x4-tap. The reason behind is that through constraints (5, 10), we strictly limit the access to registers. Other than accessing the entire register we only allow accessing the first element or the first half of the register. However as FIR-filter involves reductions over intermediate results, the hand-written code operates on elements of the same register to avoid additional latency caused by memory writebacks and double-reads. The automated code distributes the intermediate results to several registers or memories instead, incurring writeback and/or double-read latency.

Other than performance, we compared the resulting code size in Tab. 2. In all instances but one we match the code size of the hand-written code. The exception is 3x3 matrix multiplication that uses many permutation vectors for grouping 4 operations each issue cycle. The hand-written code groups them 3 by 3 instead, and avoids the need for permutations. The performances are equal since what the automation gains by grouping 4 by 4 instead of 3 by 3, it loses by permutation vector penalties. As an alternative, we tried to disallow permutation vectors completely for 3x3 to investigate the behavior of our model. The resulting code (shown as *3x3 matmul w/o PV* in the tables) had the same performance and same size as hand-written. Obviously, for applications that need permutation vectors, such an approach is infeasible.

For quantization, we match hand written code both in performance and code size.

	Exec. time (cc)		**Automation**
Kernel (|V|)	**Hand**	**Automated**	**overhead**
2x2 matmul (32)	18	18	0%
3x3 matmul (108)	36	36	0%
4x4 matmul (256)	54	57	6%
4x4-tap FIR-filter (67)	24	26	8%
8x4-tap FIR-filter (127)	29	36	24%
Quantization (256)	26	26	0%

Table 1: Comparison of generated code from Automation and hand-written assembly (cycle count).

7 Conclusions and Future work

Architectures such as ePUMA focus on providing highly efficient hardware solutions to compute-intense problems such as DSP applications. This is achieved by designing a very complex and efficient architecture with matching data access capabilities to keep the processor busy. However with the focus only on hardware efficiency, programmability tends to be forgotten. The more complex the architecture gets, the less useful traditional compiler techniques become for programming

Kernel (|V|)	Code size (bytes) Hand	Automated	Automation overhead
2x2 matmul (32)	80	80	0%
3x3 matmul (108)	128	128	0%
4x4 matmul (256)	240	240	0%
4x4-tap FIR-filter (67)	64	64	0%
8x4-tap FIR-filter (127)	128	128	0%
Quantization (256)	208	208	0%

Table 2: Code size comparison (bytes).

in a higher level language than assembly. This is mainly because of the staging approach that separates instruction scheduling and memory/register allocation. In the exploratory phase of our study, we tried to emulate this approach for 4x4 matrix multiplication. This resulted in 3x slower code compared to hand-written.

Our experiments demonstrate that, with the help of the constraint programming paradigm, it is possible to use a unified method for instruction scheduling, memory/register allocation and data access modeling (necessary for permutation vector calculation) for several DSP kernels. We showed that we can let a programmer that is not familiar with the details of the architecture write a kernel in a high level language and using our method, generate code that is almost as good as hand-optimized by the architect.

As constraints are independent of each other, the architect can add or remove archtiectural constraints to explore design alternatives. One example is the behaviour of registers in this study. Since there is direct access to both the memories and registers, they can virtually replace each other in an instruction. However the access rules and latency penalties for memories and registers are very different, which makes both modeling and solving very difficult. Seeing this, the architect can enable permutation vectors for registers or make register accesses even stricter so they can replace memory accesses in even less occasions.

Even though the variation in size and structure for the applications we targeted in our experiments makes a good case for the performance of our approach, we would like to test it further with kernels of different sizes and structural properties. The data structure for the memory schedule in our approach does not scale for large kernels because its size grows quadratic to number of nodes in the IR graph. This can be addressed by partitioning the graph into pieces that are solvable in a reasonable time. We would like to investigate this further and devise efficient and effective strategies for such partitioning.

Finally we would like to model the architecture of the target platform more accurately and make use of its special structure to the most. This way we can eliminate performance degradation similar to what we encountered for FIR-filter in our experiments.

References

[1] Andréas Karlsson, Joar Sohl, and Dake Liu. "ePUMA: A Processor Architecture for Future DSP". In: *International Conference on Digital Signal Processing (DSP), Singapore, 21-24 July, 2015*. 2015.

[2] Chris Lattner and Vikram 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.

[3] Gabriel Santin Hjort Blindell. *Survey on Instruction Selection: An Extensive and Modern Literature Study*. Tech. rep. ISBN: 978-91-7501-898-0. Stockholm, Sweden: KTH Royal Institute of Technology, Oct. 2013.

[4] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. "Efficient SIMD code generation for runtime alignment and length conversion". In: *International Symposium on Code Generation and Optimization*. Mar. 2005, pp. 153–164.

[5] Seonggun Kim and Hwansoo Han. "Efficient SIMD Code Generation for Irregular Kernels". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 55–64.

[6] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. "When Polyhedral Transformations Meet SIMD Code Generation". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 127–138.

[7] Dustin Feld, Thomas Soddemann, Michael Jünger, and Sven Mallach. "Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation". In: Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques, 2013, pp. 45–54.

[8] Andrzej Bednarski and Christoph Kessler. "Integer Linear Programming versus Dynamic Programming for Optimal Integrated VLIW Code Generation". In: *12th Int. Workshop on Compilers for Parallel Computers*. 2006.

[9] Peter van Beek and Kent Wilken. "Fast optimal instruction scheduling for single-issue processors with arbitrary latencies". In: *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*. Paphos, Cyprus, November, 2001, pp. 625–639.

[10] Robert Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. "Combinatorial Spill Code Optimization and Ultimate Coalescing". In: *ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES' 14. Edinburgh, UK, June 2014.

[11] Abid M. Malik, Jim McInnes, and Peter van Beek. *Optimal basic block instruction scheduling for multiple-issue processors using constraint programming*. Tech. rep. In: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence, 2005.

[12] Lukasz Domagała, Duco van Amstel, Fabrice Rastello, and Ponuswamy Sadayappan. "Register Allocation and Promotion Through Combined Instruction Scheduling and Loop Unrolling". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: ACM, 2016, pp. 143–151.

[13] Mattias V. Eriksson and Christoph W. Kessler. "Integrated Modulo Scheduling for Clustered VLIW Architectures". English. In: *High Performance Embedded Architectures and Compilers*. Ed. by André Seznec, Joel Emer, Michael O'Boyle, Margaret Martonosi, and Theo Ungerer. Vol. 5409. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 65–79.

[14] Monica Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 318–328.

[15] Mehmet Ali Arslan, Krzysztof Kuchcinski, Flavius Gruian, and Yangxurui Liu. "Programming Support for Reconfigurable Custom Vector Architectures". In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '15. San Francisco, California: ACM, 2015, pp. 49–57.

[16] Mehmet Ali Arslan and Krzysztof Kuchcinski. "Instruction selection and scheduling for DSP kernels". In: *Microprocessors and Microsystems* 38.8 (2014), pp. 803–813.

[17] Krzysztof Kuchcinski. "Constraints-Driven Scheduling and Resource Assignment". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 8.3 (July 2003), pp. 355–383.

[18] Willem-Jan van Hoeve and Irit Katriel. "Handbook of Constraint Programming". In: Foundations of Artificial Intelligence. Elsevier Science, 2006. Chap. Global Constraints.

[19] Abderrahmane Aggoun and Nicolas Beldiceanu. "Extending chip in order to solve complex scheduling and placement problems". In: *Mathematical and Computer Modelling* 17.7 (1993), pp. 57–73.

[20] Nicolas Beldiceanu and Evelyne Contejean. "Introducing global constraints in CHIP". In: *Journal of Mathematical and Computer Modelling* 20.12 (1994), pp. 97–123.

[21] Gilles Pesant. "A Regular Language Membership Constraint for Finite Sequences of Variables". In: *Principles and Practice of Constraint Programming – CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27 -October 1, 2004. Proceedings*. Ed. by Mark Wallace. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 482–495.

[22] Chenxin Zhang, Liang Liu, and Viktor Öwall. "Mapping channel estimation and MIMO detection in LTE-advanced on a reconfigurable cell array". In: *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*. May 2012, pp. 1799–1802.