HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Spécialité : Informatique

par

Steven Derrien

Plateformes, méthodologies et outils pour la conception d'architectures matérielles reconfigurables

à soutenir le 13 décembre 2011 devant le jury composé de :

M. Tanguy Risset, Professeur INSA de Lyon, CITI/INRIA

M. Olivier Sentieys, Professeur Université de Rennes 1, IRISA/INRIA

M. Patrice Quinton, Professeur ENS Cachan, IRISA/INRIA

M. Jean-Philippe Diguet, Directeur de recherche CNRS, LasSticc-Lorient

M. Sanjay Rajopadhye, Professeur, Colorado State University, USA

et au vu des rapports de :

M. Pierre Boulet, Professeur Université Lille 1, LIFL/INRIA

M. Jurgen Teich, Professor University of Erlangen, Allemagne

M. Florent de Dinechin, Maître de conférence HDR, ENS Lyon, LIP/INRIA

Contents

Chapter 1

Introduction

This manuscript summarizes my research activities since the defense of my PhD in December 2002. This work has been carried in several contexts, first as a post-doc at Leiden University in the group of Ed Deprettere in 2003, then as an associate professor at University of Rennes 1, in the R2D2 research group of IRISA, and then later in the CAIRN-INRIA/IRISA project-team. Since September 2009, I am benefiting from an INRIA "délégation", which helped me initiating many new research projects and collaborations, and also allowed me to start supervising some PhD students. This chapter is organized as follows, I will first start by presenting the context in which my research takes place. This will be followed by a summary of the contributions presented in the document and by a description of the organization of the manuscript.

1.1. Context of the work

The processor and computer design landscape is facing the toughest challenge it had to confront since the introduction of the Intel 4004 processor in 1971. The reason behind this upheaval stands in two words: *parallelism* and *energy*.

The days where software performance would grow just by magic, simply by benefiting from processor clock speed (or micro-architectural) improvement are over, as the "clock speed race" ultimately hit the "power dissipation wall". With the resulting outbreak of multi-core (and soon many-core) processors, the way we have been designing software (and the tools that we use for doing it) for more than 20 years needs to be completely reconsidered.

It would be a mistake to consider that this threat is only geared at general purpose computing systems, as embedded system designers are also confronted to similar problems. They however face an additional challenge: because of tight power, cost and performance constraints, most embedded platforms have to be based on *heterogeneous* multi-core systems. Such *heterogeneous* platforms integrate, within a single die, tens, if not hundreds of computing cores, these cores being significantly different in nature (programmable/extensible processors, custom hardware Intellectual Property blocks, coarse grain reconfigurable co-processors, etc). As a consequence, efficiently designing and programming such systems poses even greater challenges.

In the mean time, ubiquitous computing is becoming a reality, probably at a faster pace than we realize it. We will soon be completely surrounded by tiny computing devices, whose role will be to sense/watch and control our daily environment in a collaborative way. In addition to the numerous ethical and legal issues that such devices may raise, they will also pose stringent new challenges to computer and system architects. They will be expected to work autonomously for months if not years, either by relying on extremely limited power supplies, or by harvesting energy from their immediate environment. State of the art energy harvesting can provide at most $20\mu W$ whereas the power budget of a low-power MCU such as the MSP430 is around $5\,mW$ at $16\,MHz$. Satisfying these requirements means filling a two orders of magnitude gap in terms of energy efficiency improvements. Here again, radically new approaches are needed.

Addressing these challenges requires combining techniques and knowledge from several fields from Electrical Engineering (CAD tools, digital design, micro-electronics) to Computer Science (processor architecture, operating systems, compiler design, software engineering). In such a context, this work has been revolving around the definition of tools to help the design of complex

heterogeneous systems on chip. More precisely, this work has been focusing on automatic synthesis of hardware co-processors, to improve the energy efficiency and performance of embedded platforms. Using custom accelerators has shown to bring significant performance and energy improvements (from $\times 10$ to $\times 50$), however their design is very tedious and time consuming as it is done through *Hardware Description Languages* such as VHDL or Verilog.

Silicon compilers that can generate custom hardware from a high-level language, have been advocated as an answer to address this design time issue. Even though the first of such tools date back to the early 90s, it is only recently that they finally found their way in real design flows, as their efficiency and usability remained limited for quite some time. Today, there exists a large choice of robust and mature C to hardware tools [73, 30] that are used as production tools by world-class chip vendor companies.

However, there is still room for improvement, as these tools are far from producing designs with performance comparable to those of expert designers. The reason of this difference lies in the difficulty, for automatic tools, to recover information that may have been lost during the compilation process. The difficulty also stems from the lack of well defined target, as these tools must jointly define the hardware architecture and map the algorithm onto that architecture. This lead to some subtle trade-off between performance, resource usage, and energy efficiency. Our contributions in this field focused on studying how to extract and take advantage of massive parallelism and to improve energy efficiency of circuits designed using such silicon compilers.

We also investigated direct applications of reconfigurable computing, by proposing and designing several massively parallel application specific accelerators for multimedia database search and bioinformatics. These two fields involve many extremely computationally demanding algorithms that are well suited to hardware acceleration. In particular we have studied how to accelerate Markov based profile search, and specifically the HMMER software suite, a reference tool for the whole bioinformatics community [63].

We also studied the hardware acceleration of content-based retrieval algorithm for multimedia data, and in particular for image databases. These algorithms, whose goal is to search the nearest neighbors of high dimensional vectors, suffer from what is known as "the curse of dimensionality" and a search hence requires an exhaustive scan in the reference database. Here again, we have shown that these applications were good fit for hardware acceleration.

Lastly, in the context of a close collaboration with Dominique Lavenier from the IRISA/INRIA Symbiose team, we also contributed to the design and proposal of reconfigurable hardware based on the concept of smart storage systems, targeted at searching information in large database systems. This research work led to the construction of two prototypes: RDISK and ReMIX.

It is noteworthy that these three topics mentioned above are tightly related to each other, and addressing all these issues at once is unavoidable. As a matter of fact, the ability to improve design tools requires two things: a deep understanding of hardware platforms and FPGA technology, but also a deep knowledge of applications requirements so as to help pinpointing the limitation of existing tools. Experience has shown that such an expertise can only be acquired through recurring practical case studies on real life target platforms. Figure 1.1 illustrates the contributions presented in this work, by showing how they are related to the aforementioned topics.

1.2. Summary of the contributions

This section provides an overview of the research topics presented in this document. These topics are presented in chronological order, starting from earlier work which took place during my post-doc and/or that were in the direct continuation of my PhD work, to some very recent

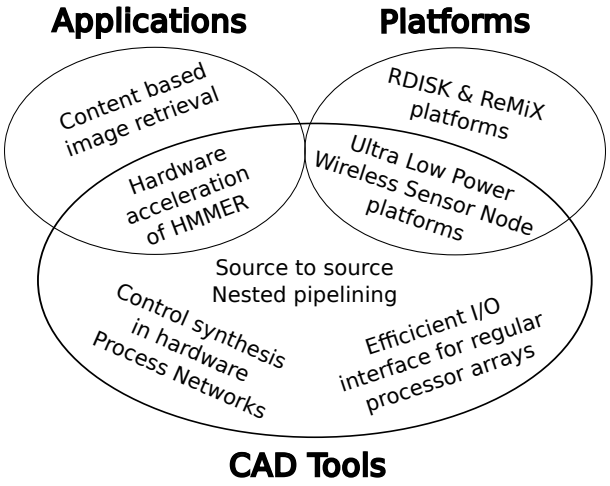


Figure 1.1.: A visual representation of the contributions presented in this work.

work on source-to-source transformations for High-Level-Synthesis (HLS).

1.2.1. Synthesis of loop nests hardware accelerators

The first contribution after my PhD was a result of a stay as a post-doc in 2003 at Leiden University in the group of Ed Deprettere, where we worked on the problem of automatic hardware synthesis in the context of the Compaan/Laura framework [165, 153]. The goal was to synthesize efficient hardware process networks [18] leveraging the Polyhedral Process Network semantics [154]. To address the problem, we studied and investigated several approaches to improve the scalability and efficiency of the hardware controllers generated from Compaan.

After my returning to University of Rennes, I have been working with Tanguy Risset and Alain Darte on the problem of automatic synthesis of efficient hardware/software interface for processor arrays, in the context of their integration as hardware IPs in a System on Chip. Some of my earlier work with Tanguy Risset addressed the problem of interface synthesis for one dimensional arrays, integrated as slave peripherals in a system. In this work, we extended our previous results to support partitioned processor arrays with busmaster capabilities. In particular, we proposed a technique, which is able to determine a static I/O pipeline scheme for distributing the data among the boundary processors of the array, given a partitioning of the array enforcing certain constraints,

1.2.2. Reconfigurable platforms for high performance computing

In the mean time, I also continued a collaboration started with Dominique Lavenier just before the end of my PhD. Our joint work has focused in the design and utilization of application specific reconfigurable platforms for high performance database processing. Two prototypes have been designed during this period, namely the RDISK and ReMIX machines. The aim of the RDISK project was to propose a new paradigm of reconfigurable hardware accelerators based on the *smart-disk* principle [87, 131, 110], the goal being to move the computational resource as close as possible to the data source. Our key idea was to attach the reconfigurable accelerator directly to the disk output, so as to be able to perform on-the-fly data filtering. A prototype RDISK cluster with 48 nodes was designed, and several bioinformatic applications (BLAST [36], WAPAM) have been accelerated on the platform, with speedup ranging from 5 to 20 per node. The RDISK cluster platform has also been serving as a reference target platform

for three PhD students (S. Guyetant, M. Giraud et A. Noumsi), and the cluster was later added to the processing resource of the GenOuest platform (`http://www.genouest.org`).

The ReMIX platform was in continuation of the RDISK, with a focus on large scale indexing, where index size exceeds capabilities of current volatile memory technologies. The idea consisted in coupling the FPGA resource to Nand Flash memory. Each node of the ReMIX cluster hence contains a Xilinx FPGA combined with 64 GB of Flash memory, organized in parallel banks to maximize I/O throughput. The approach permits to combine the benefits of non permanent storage system with those of random access memories (throughput, access latency). An 8-board prototype was realized and validated, and several accelerators have been ported to the system, with speedup factors of up to $\times 50$ per node.

1.2.3. Application specific parallel hardware accelerators

While working on the RDISK and ReMIX platforms, we have also conducted several case studies. The goal of this work was twofold. The first motivation was to demonstrate the relevance of FPGAs for high-performance database applications. The other one was to see how semi-formal design methodologies could help deriving more efficient accelerator architectures, but also to search for interesting design tricks/optimizations that could be worth being integrated in high-level synthesis tools. The target application domains in this work were content based search in multimedia databases (image, video) and genomic sequence comparison in bioinformatics.

In the context of the PhD of August Noumsi (supervised by Patrice Quinton), we have been working in collaboration with the IRISA/INRIA TexMex group on the hardware acceleration of content based image retrieval algorithms. A detailed description of the work carried on this topic is provided in Section 2.2. In particular, we have quantified the impact of some architectural parameters (encoding accuracy, synchronization overhead) on the effective performance of two hardware implementations (for the RDISK and ReMIX platforms), which have exhibited important (more than $\times 100$) performance improvements.

The other application domain we have studied in this context is bioinformatics, and more particularly genetic sequence comparison algorithms. The contributions related to this work are detailed in Chapter 3. These compute intensive kernels, based on dynamic programming algorithms, are very good candidates for hardware acceleration. In addition, the fact that genomic sequence databases grow in size at a faster pace than processor performance raises a need for high performance implementation of the algorithms. A number of such algorithms and applications have already been successfully accelerated on FPGAs (Smith-Waterman [140], BLAST [36]) with good performance improvements (speedup by $\times 100$ over a software implementation). Some other algorithms turn out to be more difficult to accelerate in hardware, in particular when there is no obvious parallelization scheme. This is the case for the HMMER software suite, based on Markov models, used for sequence similarity searching.

We have proposed a scalable parallelization scheme for this algorithm, detailed in Section 3.2, which is well suited to hardware acceleration. This scheme is based on space-time transformations formalized in the polyhedral model, and enables the derivation of a generic processor array template that can easily be retargeted to different accelerator platforms, while retaining its efficiency. The template is parameterized and supports different levels of fine grain pipelining, and can handle resource constraints by allowing the designer to specify the number of processing elements used in the architecture. This result was presented in 2007 to the *IEEE International Conference on Application-specific Systems, Architectures and Processors* and received the best paper award.

We continued investigating this topic in the context of the PhD of Naeem Abbas (funded by the ANR BioWic project) and in close collaboration with Sanjay Rajopadhye from Colorado

State University. Our work focused on sophisticated program transformations leveraging algebraic properties of the algorithms which extensively use reduction operations. We have shown that to the contrary of common beliefs, it is possible to parallelize the HMMER main algorithm at the price of a moderate increase in computation volume. The idea, presented in Section 3.3, is based on the uncovering of a hidden parallel prefix operation, for which there exists many efficient parallel hardware implementations, as the topic is studied since the mid-70s. The parallelization of a prefix operation however comes at the price of a slight increase in computation workload ($O(N.\log_2 N)$ instead of $O(N)$). The approach was validated on a high performance FPGA accelerator (XD2000i) and demonstrated speedup by up to factors of 7.5 over the highly optimized software implementation on a 3 GHz Intel Dual Core. We are currently pursuing the automation of some of these transformations within a C to hardware compilation tool.

1.2.4. Tools for application specific architecture design

In 2007, I decided to refocus my research activities on CAD tools, and to start working again on automatic synthesis of parallel hardware accelerators. This decision was motivated by the work of Ludovic L'Hours on the first version of the Gecos compiler [105], which opened new interesting perspectives. Since 2008, Gecos serves as the reference compiler infrastructure for my research activites and for the whole CAIRN group. Two different problems have been targeted: the synthesis of ultra low-power controllers for wireless sensor networks, and source-to-source loop transformations for HLS.

Ultra low power wireless sensor node platforms

Wireless Sensor Networks (WSN) are a very promising technology with potential applications in many domains of daily-life, such as structural-health and environmental monitoring, medicine, military surveillance, robotic explorations, and so on. A WSN is composed of a large number of nodes deployed inside a region of interest. Designing a WSN node is challenging, as designers must deal with strong form factor and energy constraints. For example, WSN nodes may need to run autonomously for months using limited energy sources (between ten and a few hundred mAh for small form factor Lithium batteries), or by harvesting energy from their environment (power supply of at most $20\,\mu W$).

Current WSN nodes are based on micro-controllers such as the MSP430 [150] with typical power dissipation of a few mW, which are still orders of magnitude too high for many candidate applications of WSN. In the context of the PhD of Adeel Pasha (graduated on December 15 th 2010) and in collaboration with Olivier Sentieys, we tackled the problem by using a radically different approach, combining hardware specialization with concurrent task-level processing. The idea was to take maximum advantage of the opportunities of specialization by generating, for each task of the system, its custom hardware controller. By combining this idea of specialization with *power gating*, we have shown that it is possible to significantly reduce dynamic power while maintaining very low levels of static power dissipation.

Our approach, detailed in Chapter 4, leverages tools and techniques from micro-electronics, compiler/hardware synthesis and software engineering to offer a complete flow. This flow enables the complete specification (and synthesis) of a hardware/software platform from a combined use of C and of a platform description Domain Specific Language. The implementation of such a complex toolchain was made possible thanks to the use of Model Driven Software Development tools, which allowed us to benefit from many facilities for code generation and DSL design provided by the *Eclipse Modeling Framework*. Very interesting experimental results have been obtained, which showed that overall energy improvement by two orders of magnitude (w.r.t. to

the MSP430) is possible through this approach. This work has led to four publications including a paper presented at the IEEE/ACM Design Automation Conference in 2010.

We are pursuing this research direction in the context of the PhD of Vivek D. Tovinakere (under the supervision of Olivier Sentieys). His focus is on accurately modeling the impact of fine grain power gating techniques, and on the introduction of hardware reconfiguration. However, this work, still in progress, will not be presented in this document.

Source-to-source transformation for HLS

High-Level Synthesis tools, despite their progress over the last few years, do still suffer from many limitations. The initial source code that is to be mapped on hardware almost always requires to be heavily modified so that the tools are able to deal with it, but also (and mostly) so that they can derive an efficient architecture out of it. Such transformations can have significant impact on performance whenever the input code involves nested loops operating on arrays, since it is often possible to rewrite the loop to expose additional parallelism or improve locality by exploiting data reuse opportunities directly at the source code level.

I have been working with Antoine Morvan (PhD student supervised by Patrice Quinton) on a source-to-source loop transformation toolbox based on polyhedral representations of loops since late 2008. This work is part of the S2S4HLS project [1] funded by the INRIA-STMicroelectronics Nano2012 program. In this work, we have been focusing on improving pipeline efficiency for loop nests that can be represented in the polyhedral model.

Efficient hardware acceleration of compute intensive kernels generally involves pipelining the execution of these kernels. This pipelining is applied to the innermost loops of the kernel and consists in overlapping the execution of several iterations on the hardware resources. The technique significantly improves the throughput of the accelerator, at the price of a slight increase in area and latency (related to the pipeline depth). It turns out that loop nests are well suited for a pipelined execution, as they often carry a significant amount of iteration level parallelism. However the efficiency of this pipelined execution degrades quickly as the number of iterations within the inner loop goes below a certain threshold. The degradation is caused by the pipeline fill/flush stages which then dominate execution time. This situation appears very frequently in the context of high-level synthesis, as many algorithms exhibit low innermost loop trip counts and since designers often try to derive deeply pipelined datapath, with long fill/flush stages.

In this work we propose to reduce this overhead by pipelining the execution of the whole loop nest instead of restricting the pipelining to the innermost loop. The idea consists in coalescing the loop nest hierarchy in to a single loop, that will be further pipelined. The key problems are then (i) to build this coalesced loop, and (ii) to make sure the pipeline of this coalesced loop is legal (i.e., that it does not alter the semantics of the original code). Ensuring the legality of such a pipeline turns out to be difficult in the general case, especially for loop nests involving non constant loop bounds.

Our contribution, detailed in Section 5.5, consists in a legality check taking advantage of iteration instance wise dataflow analysis, associated with a correction technique. This correction technique can make a pipeline legal by inserting extra dummy iterations in the coalesced loop, these iteration then serving as wait states. Our technique was implemented in the Gecos source-to-source compiler and has shown encouraging results, with performance improvement of up to 30 % for matrix multiplication and QR decomposition.

1. Source-to-Source for HLS

1.3. Organization of the manuscript

The remainder of the document is organized as follows, Chapter 2 describes our work on reconfigurable accelerators for unstructured databases, along with a case study application on multimedia content based search. Chapter 3 continues to discuss the topic of unstructured databases, with the focus on the design of a hardware accelerator for a sequence comparison algorithm (HMMER), widely used in the bioinformatics community. In Chapter 4, we present our work on ultra low power wireless sensor node platforms, and the design flow that supports the idea of hardware microtasks. Chapter 5 provides an overview of all our contributions on the topic of hardware synthesis in the polyhedral model. Chapter 6 summarizes our contributions, and provides detailed research perspectives both in the short and long term. A complete Curriculum Vitae, and a list of publications are also provided as appendix A and C.

Chapter 2

Reconfigurable Accelerators for searching unstructured databases

The work described in this chapter took place between 2003 and 2006 in close collaboration with Dominique Lavenier from the Symbiose team at IRISA. The goal was to see how reconfigurable technology could be used for accelerating search operations in unstructured databases (such as multimedia and genomic databases).

In this work, we tried to take a fresh look at reconfigurable accelerator platforms, by reconsidering the way the reconfigurable resource was coupled to the rest of the system. Our approach was motivated by our earlier experiences which showed that I/O throughput and/or latency were almost always performance bottleneck for FPGA platforms. To address this issue, we proposed two original prototype systems where the FPGA processing power is set as close as possible to the data source, following the principle of *smart storage systems*. We have also evaluated how this type of platforms could benefit to multimedia content-based search applications.

In Section 2.1, we discusses the idea of reconfigurable storage system and describe two platform prototypes RDISK and ReMIX that we designed following this idea. We then detail two different accelerators for our multimedia search algorithm. These implementations leverage both coarse grain and fine grain parallelism to reduce the search time and were carefully designed to maximize global (i.e., system level) performance. We illustrate the relevance of our approach through a case study: content-based image retrieval for large databases.

2.1. Reconfigurable platforms for database processing

The use of reconfigurable hardware for accelerating general purpose and scientific computing has been an active topic of research for now more than 20 years. In theory, reconfigurable accelerators are able to offer the *best of both worlds*: flexibility, thanks to dynamic reconfiguration, and performance thanks to parallelism and specialization. Seminal work in the domain [157, 156, 82] that took place in the early 90s had shown very promising results, and set the stage for what came to be known as the "Custom Configurable Computing" community.

However, in the late nineties, the lack of high-level design flows combined with ever increasing processor performance and severe I/O bottlenecks in FPGA platforms somewhat brought to an end most of the interest in reconfigurable computing. In particular, the steady improvements in both clock speed and micro-architectures had raised the bar too high for FPGAs to be able to compete with high-performance general purpose programmable processors. In the mid 2000s, the so-called *clock speed wall* and raising concerns for power dissipation in processors renewed interest for alternative technologies such as FPGAs. In the mean time, the outbreak of multi-core and GPUs, and in particular the programming challenges that they would represent, contributed to make FPGA design effort more acceptable, especially as many usable C to hardware compilers became available during this period. As of today, it seems that Reconfigurable Computing has found a small market niche, yet it remains to be seen whether this niche will survive in the longer term.

Independently of these architectural evolutions, another interesting phenomenon was ob-

served during the last decade: the fast wide spreading and growth of what is known as *unstructured databases*, carrying diverse types of information, from genomic to multimedia data. Because they cannot benefit from traditional indexing techniques (i.e., those used in relational databases), searching those databases often involves a complete scan of the dataset. This leads to poor performance as the whole data has to be read from its permanent storage medium but also since the selection algorithm has to be applied to every item, which (as far as genomic and multimedia data are concerned) is usually computationally expensive. Given that these databases size ranges from hundred of gigabytes to terabytes, and that their size tends to grow exponentially, running these analyses requires *large storage capabilities* along with *huge computing power*. In practice, only very few systems can provide both; supercomputers are geared towards processing power and storage systems are geared towards data distribution. Hence the user usually must sacrifice one for the other.

2.1.1. The case for Active Storage systems

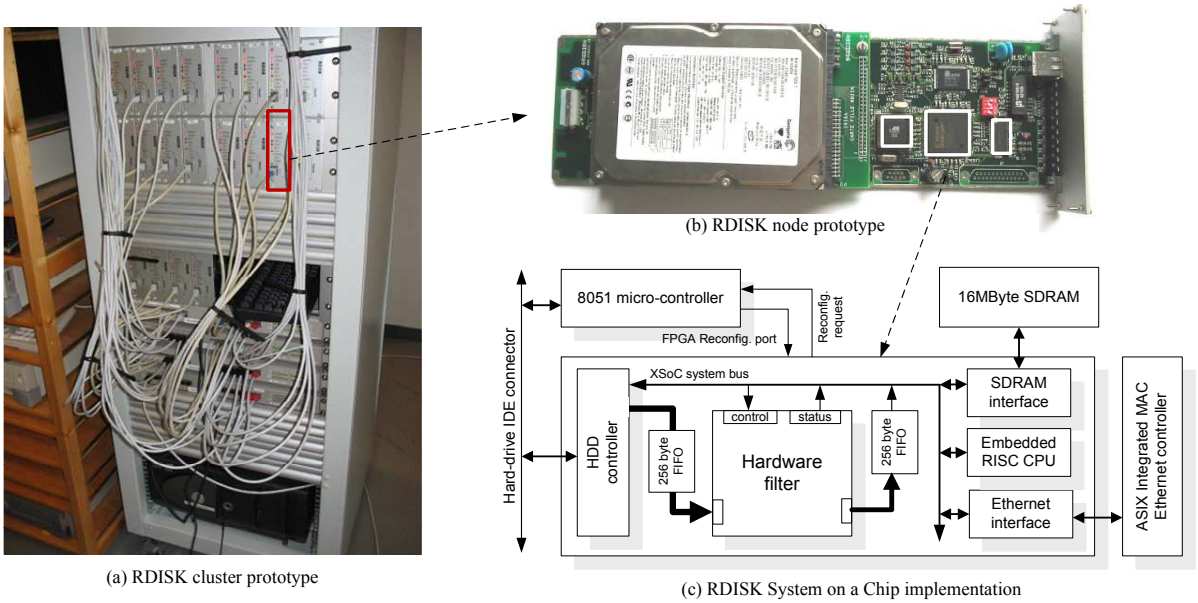
Storage bottlenecks are solved through the use of High Performance data warehouses based on *Storage Area Network* (SAN). SANs are built upon a high-speed special-purpose networks interconnecting storage devices with servers on behalf of a larger network of users. The evolution of storage systems technology has shown that more and more *intelligence* is embedded into these systems (network protocols, cryptography, file systems management, etc.). While these tasks are still mainly geared toward data management and availability, we believe that the growing gap between the storage and processing capacity might induce a shift in the way these systems are designed. In particular, we believe that they may start providing data processing oriented services in the future. There are two reasons for this:

- In most cases the end user is only concerned with a small fraction of the original dataset that contains relevant information to its query. This is typically the case in DNA sequence similarity search and in multimedia content based retrieval problems. Filtering the data at its source has many benefits, as it significantly lowers the I/O volume in the storage system, hence decreasing its power consumption, and also improving its scalability.
- For such products, the cost overhead induced by additional embedded processing power would remain small (many SAN infrastructures already embed FPGAs for I/O management) with significant outcomes in terms of performance and energy.

Bringing computational power closer to the data source is not a new idea, and was explored in the context of *smart disk* devices. The *smart disk* approach suggests to use a fraction of computing power available within the hard-drive embedded controllers for data processing applications, based on the information that these processors, whose role is limited to disk head scheduling or cryptography/security, are somewhat underutilized. This concept has been investigated by several research groups in the late 90s [87, 131, 110], and their work showed promising results for a relatively wide panel of applications in which large datasets were involved. However it appeared that the hard disk storage industry would not move toward such open embedded CPU based smart disk devices, mostly because of security [1] and cost reasons. This is not true for the *storage systems industry* which is based on more open protocols and infrastructures.

Following this vision, we propose a new type of storage system based on "networked intelligent storage devices", which could be assembled in order to build larger scale application specific database systems. The *intelligence* of our system would lie in its ability to perform *on-the-fly filtering* of the data as it is read from the storage device. The goal is to send a small superset of the relevant data to the host. This superset will then be further analyzed to filter out

1. The software IP in hard-disk controllers is a key contributor to the actual hard disk drive performance, and this expertise is highly protected.



(a) RDISK cluster prototype

(b) RDISK node prototype

(c) RDISK System on a Chip implementation

Figure 2.1.: The RDISK Node cluster and node prototypes, along with the node reconfigurable System On Programmable Chip

false positives. Because the performance benefits of a CPU based smart disk is not expected to justify its cost overhead and design complexity, the only way to be competitive is to use hardware acceleration through reconfigurable technology.

In the following, we summarize the work (carried in close collaboration with Dominique Lavenier) on RDISK and ReMIX , two intelligent storage prototypes designed between 2003 and 2006.

2.1.2. The RDISK platform

The first research project that we started aimed at proposing a smart disk platform leveraging reconfigurable technology, and following the principles of Network Attached Storage (NAS): a computer data storage device connected to a network that provides filesystem level services. The following provides an overview of the work done on the RDISK platform design and implementation, both from the architectural and system level points of view.

One of the goal was to propose a low-cost and scalable system, competitive in prices (in 2003) with other Network Attached Storage devices. We therefore targeted a cost of roughly $200 per RDISK node, including PCB and a middle range HDD. This led to the choice of a platform based on a low cost $30 Xilinx SpartanII-200 FPGA coupled to a 100 Mbit controller with 32 MB SDRAM memory and a 8051 controller for handling the device configuration. The core of the RDISK system was a *System on a Chip* implemented on the FPGA, represented in Figure 2.1.(c). Because of stringent resource constraints, we based the system on an open source 16 bit RISC soft-core processor, that requires less than 300 logic cells, leaving 2500 LUTs and 10 BlockRams for the data processing filter. This core was in charge of the network management and implemented a custom light weight UDP protocol. To maximize throughput, the hard-drive controller was implemented as an advanced autonomous state-machine that could handle most of the IDE-ATA protocol, and perform complex I/O requests. Our file system services were handled by the host server, to ease development.

To demonstrate the feasibility of our approach, we built a complete parallel storage system, by assembling a cluster of 48 RDISK nodes together, as shown in Figure 2.1.(a). The abstract system-level view of this RDISK cluster is provided in Figure 2.2. In this system, all disks can operate in parallel, and their results are sent to the host through the network. The aggregated RDISK nodes output throughput must not saturate the host network link to guarantee the absence of bottleneck. This puts a constraint on our hardware filter selectivity (the less selective the filter is the more data has to be sent to the host). As an example, our prototype uses 48 boards, we assume that the host can handle at most 5 MB/s (half the peak 100 MBits), and that each drive provides a sustained bandwidth of 15 MB/s. This means that our hardware filters should be able to filter out 99.3% of the data read from the disk, with each individual disk output bandwidth being in the range of 100 kB/s.

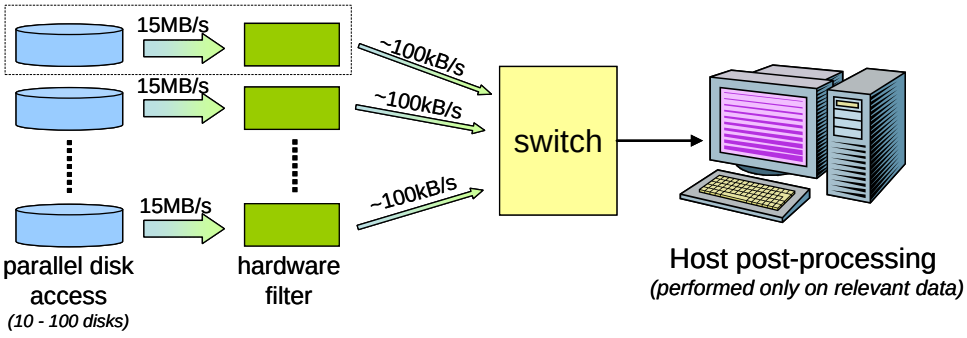


Figure 2.2.: The RDISK cluster: system level view

Since the average network throughput requirements for each node are in the 100kB/s range, we did not try to optimize our network firmware and chose an IP/UDP stack in order to simplify the development of the network management layer. To minimize overhead, we used a simple UDP based protocol to receive and send commands to and from the host machine. It is worth noticing that most of this protocol could have been implemented in hardware using a complex state machine similar to the one used for controlling the disk. It would have then been possible to perform I/O access using the full 100 Mbits Ethernet bandwidth.

An important feature of the RDISK system is the way it handles *dynamic reconfiguration management*. All configurations for a given RDISK board are stored locally on its HDD, within a specific partition of the file-system. These bitstreams are accessible to the 8051 microcontroller that serves as a *configuration manager* and shares access to the HDD with the FPGA. When a new hardware configuration is to be run on the system, the host simply sends a *suicide* command to the selected boards, including the next configuration to be used. This message is forwarded to the configuration manager, which reconfigures the FPGA using the corresponding bitstream read from the disk, and loads its associated OS image (the whole operation lasts 800 ms).

The RDISK served as a target platform for three PhDs (S. Guyetant, M. Giraud and A. Noumsi). In 2006, the prototype was integrated to the pool of computing resources provided by the Génouest platform [2]. Several bioinformatic applications have been successfully ported to the architecture [3], including a hardware accelerated version of the BLAST sequence alignment by S. Guyetant and a similarity search tool based on Weighted Finite Automaton (WAPAM) by Mathieu Giraud. In particular, the WAPAM program leveraged the dynamic reconfiguration capabilities: the application would synthesize on the fly a custom hardware design tailored to

2. `http://www.genouest.org`
3. I was not directly involved in these contributions

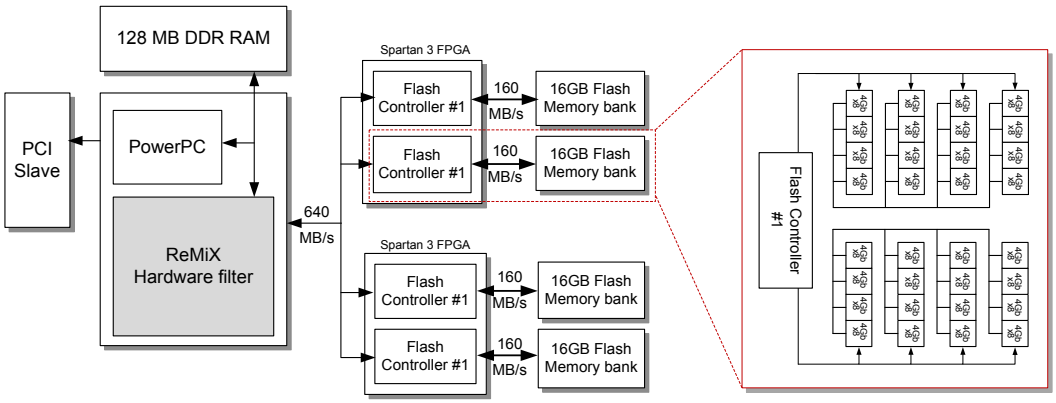


Figure 2.3.: Simplified RMEM node architecture

the pattern at hand, and would then configure the platform with the problem instance specific accelerator. This tool was later made publicly available to registered platform users from the bioinformatics community. Even though I was not involved in these contributions, they show the relevance and interest of our proof of concept.

Another application field that was explored in the context of RDISK is content-based multimedia search. This topic is thoroughly described in Section 2.2, and will not be detailed here.

I took an important part in the design of the RDISK architecture, which was the core of S. Guyetant PhD thesis, as the node architecture was mostly based on one of my draft design. This work was carried in 2002 and 2003, before that soft-core processors (and their associated tools) would become commodity components. The development of the RDISK SoC hence required a significant development effort, to which I also contributed a lot (in particular on the soft-core CPU and on the SDRAM memory and cache controllers).

2.1.3. The ReMIX Platform

The ReMIX project was started in 2005, in continuation of the RDISK project, in the context of a *Action Concertée Incitative* proposal. The design of ReMIX was mainly motivated by two observations:

- Even if attached directly at the disk output, I/O bandwidth and/or random access time still remained to be a bottleneck and prevented from fully utilizing the power of the FPGA.
- Nand-Flash memory did not suffer from the same limitations and its price had been decreasing at an exponential rate. We expected that this technology would soon become a serious competitor to magnetic storage systems.

The idea behind ReMIX was to benefit simultaneously from the high data throughput and the short access-time that can be obtained with the parallel use of Flash memory devices, and from the high computing density of a high-end FPGA. The goal was to obtain the largest possible storage capability and the smallest possible random access time.

Nand-Flash memory exhibits significant differences with respect to standard memory in the way data is accessed. In Flash technology, data is addressed at the *page* level, each page containing between 512 byte and 2 KB. Each access suffers from a relatively long latency ($20\mu s$), but which remains three orders of magnitude better than a hard disk drive. The main weakness of this technology is the relatively slow write operation, which requires a page to be erased before it is rewritten, and the fact that the number of erase operations for a page is limited. This puts an upper bound on the lifetime of the device, and forces designers to resort to advanced *wear*

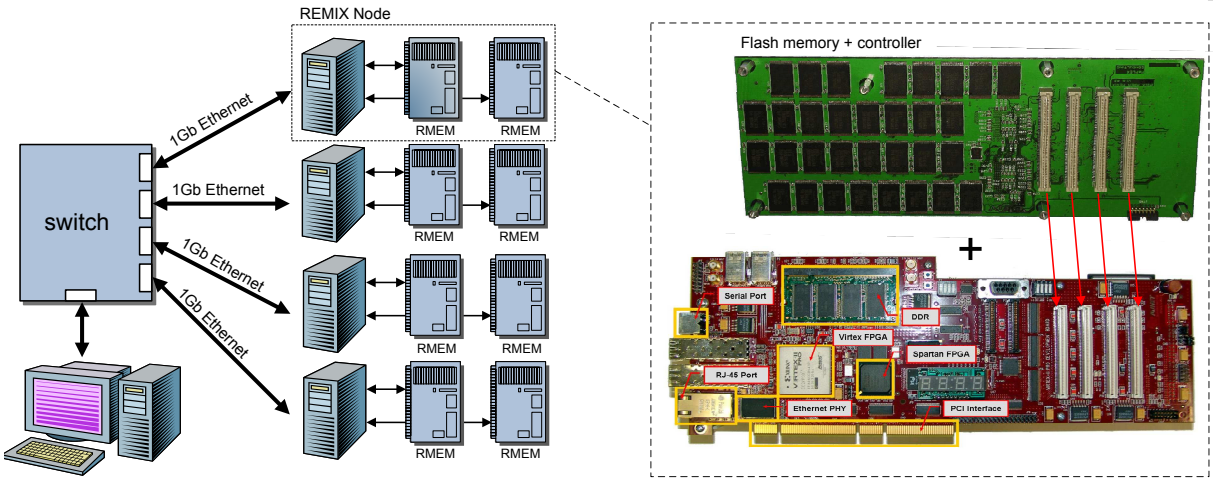


Figure 2.4.: ReMIX cluster architecture

leveling strategies as in modern Solid State Drives.

To minimize the hardware development effort, we based our design on a pre-existing third party PCI board integrating a Xilinx Virtex-II Pro FPGA. We then designed an extension board associating 64 GB of Nand-Flash memory [4] and several smaller FPGAs for handling the complex Nand-Flash protocol. In our system, the memory is organized in multiple parallel banks as illustrated in Figure 2.3, enabling sustained I/O throughput of 640 MB/s. A simple host managed file system was used to allow the user to transparently manage the content of the 64 GB Flashmemory, while guaranteeing maximal throughput when accessing data during processing stage. One weakness of this platform is the poor filter output throughput, which is restricted to 5 MB/s (the target PCI board only supporting slave I/O). A prototype cluster of 4 ReMIX node (with two ReMIX board per node) was designed and validated. The cluster, represented in Figure 2.4 was used in several other projects [98], including a port of our Multimedia application [3], that is described in the following section.

2.2. A case study: Multimedia Content-Based Image Retrieval

Content-Based Image Retrieval (CBIR) is a technique that is used to retrieve images from a database that are (at least) partly similar to a given query image. CBIR is drawing increasing interest due to its potential application to problems such as image and video copyright enforcement. Indeed, the large use of the Internet resulted in a huge increase of multimedia content available on the Web. Enforcing copyright has hence become a concern to its owners, and in particularly identifying undue use of images is an important issue. Because Internet is a rapidly changing support, there is a need for precise and fast image comparison algorithms that could be used to inspect the web on a daily basis.

CBIR is mainly based on comparison of *image descriptors* of a *query image* with those of *database images*. Descriptors may be either *global*, i.e., they represent some global feature of an image (e.g., a grey level histogram) or *local*, in which case they describe special points of interest in the image (e.g., corners, color changes, etc.) as depicted in Figure 2.5. In this work, we deal with local descriptors as they were shown to be more robust since they are less dependent to image variations than global descriptors [122].

4. In 2005, time of the design, only 32 Gb Nand-Flash chips were available)

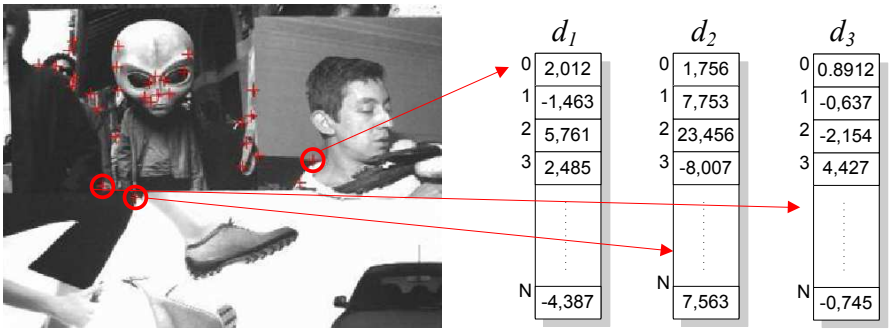


Figure 2.5.: Local descriptors are constructed from interest points of the image

2.2.1. Similarity search algorithm

Retrieving an image is done in three steps. The first step consists in associating a set of descriptors for the query image, typically, a few hundred vectors of 24 (or more) real components. The second step then computes the distance between each one of the query descriptors and those of the image database (we call this step the *distance calculation stage*). In the third step, we construct, for each query descriptor, a k-nearest neighbor list which contains the k database descriptors with the smallest distance to the query descriptor (*selection stage*). Finally, votes are assigned to the images depending on their appearance in the k-nearest neighbor lists (*election stage*): the image that has the largest number of votes is considered to be the best match.

The whole process is extremely time consuming: retrieving an image among a 30,000 image database required about 1,500 seconds on a standard workstation (in 2005). This is obviously impractical for many applications that would require low response time. The reason for this inefficiency was [5] the lack of efficient indexing scheme for such high-dimensional space, caused by what is known as the *curse of dimensionality* [37]. One solution to improve the performance of CBIR systems was to resort to special purpose implementations. Parallelizing the application on a parallel machine was the most natural choice, and has already been studied by a few authors, including Robles et al. [45, 133].

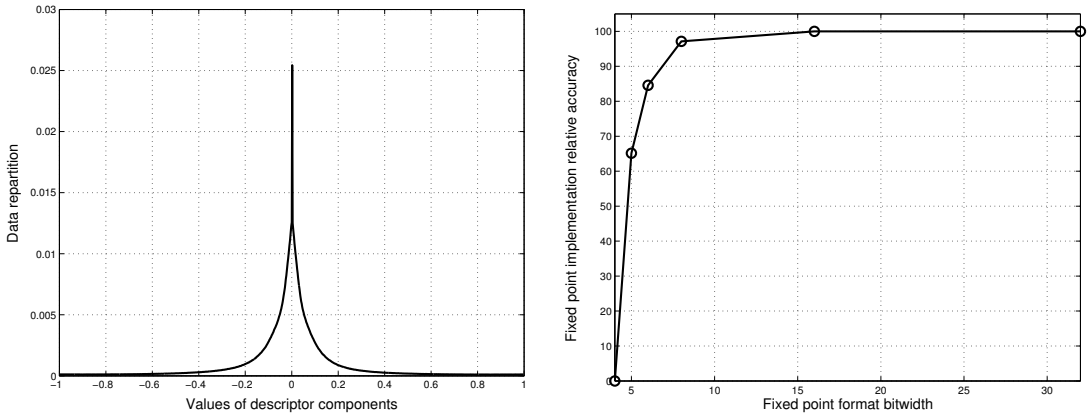
In this work, we approached the problem by combining both parallelism and special-purpose hardware. Our target architectures were the RDISK and ReMIX Platforms, and we therefore proposed two very different hardware accelerators for the algorithm. Rather than proposing ad-hoc solutions, we tried as much as possible to guide our design choice using high-level analytical performance models. The goal was to be able to take into consideration many different constraints: FPGA resource constraints, bandwidth constraints, and also some statistical information on the algorithm behavior (numerical value distribution, and run time profiling). These models would be too complex to describe in details and we invite the reader to refer to our publications on the topic [23, 3, 29].

2.2.2. Adapting the algorithm for hardware acceleration

The first step in porting the algorithm to hardware is to see whether it can be transformed in order to make it more amenable to custom hardware acceleration. In the following, we describe two of such transformations that we used in this work.

The first one is a conversion from floating-point encoding to short fixed-point integer arithmetic. Even though floating-point operators can be implemented in programmable logic [57, 56],

5. The use of past tense is deliberate, as this fact does not hold anymore, as explained in the last section of the chapter.



(a) Descriptor values distribution (b) Accuracy as a function of encoding

Figure 2.6.: Descriptor values distribution search accuracy

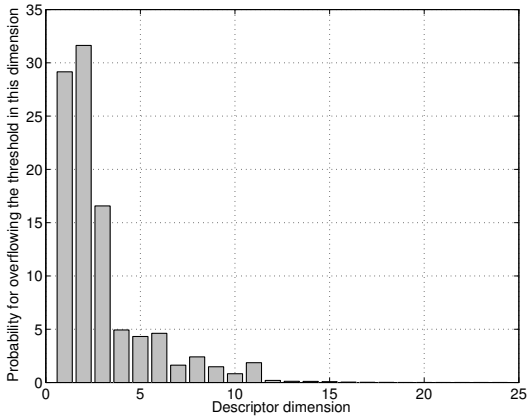
their area efficiency and performance are still very far from that of fixed-point integer arithmetic. Devising a custom (and short) fixed-point representation for the algorithm data has several advantage: it improves the circuit clock speed, helps increasing the degree of parallelism in the accelerator (more operators can be implemented for a same resource cost), and significantly improves database scanning time, as the database size can also be reduced. Figure 2.6(a) shows the distribution of CBIR descriptor values. The reader may notice that the range of these values is concentrated within a narrow interval. This suggests that descriptor data is amenable to short fixed-point encoding.

The second transformation was the choice of an alternative to the Euclidean distance metric used to compute the distance score. In particular, a common substitute for the Euclidean (also known as L_2) distance, is the Sum of Absolute Difference (SAD) distance (also known as L_1) metric, which is more suited to hardware, and also widely used in image processing.

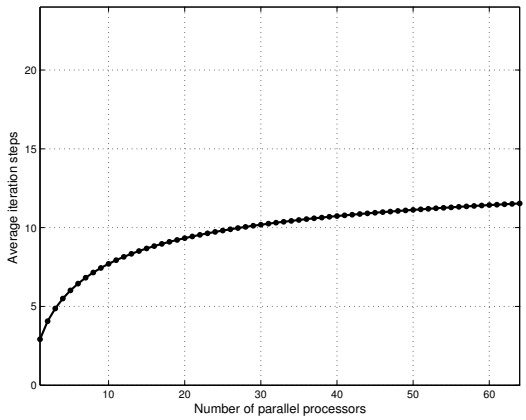
In the context of CBIR, there is no way to directly model the impact of a loss of accuracy (by calculating the SQNR) of the search results. The only solution is to use extensive simulation and to experimentally observe the impact of a given conversion scheme (scaling factor and bitwidth) on the search results. This validation was realized through an accuracy test based on the work of Amsaleg et al. [37]. This test considers a random image I_{ref} taken from the image database. From this image, it derives a set of image variations (including I_{ref}) using a set of transformations (cropping, rotation, JPEG encoding, etc.) taken from the Stirmark benchmark [122]. Each one of these images is then used as a query for the database to evaluate the accuracy of the retrieval. Experimental results for various fixed-point bitwidth format using L_1 distance are given in Figure 2.6(b). They have been obtained for a large set of images, and correspond to several week of running time. They show that for 8 bits fixed-point encoding and above, accuracy is similar to the original software (whose accuracy is 85%).

The original software implementation of the algorithm took advantage of a straight-forward, but very efficient optimization. Whenever the current distance score between a query vector and a database vector exceeds the score of the k^{th}-nearest neighbor (in the following, we call *threshold* this value), the later is sure to never appear in the list. It is therefore useless to continue, and the algorithm can directly proceed to the next candidate vector.

Figure 2.7(a) illustrates the efficiency of this *early exit* approach, as it shows that in 80 % of the cases, only three iterations are enough to discard a candidate vector. However this only translates into a moderate speedup [23], mostly because of micro-architectural side-effects (the optimization induces many incorrect branch predictions). Taking advantage of this optimization



(a) Probability of early as a function of the iteration number



(b) Average number of iterations as a function of the number of processors

Figure 2.7.: Efficiency of the early exit optimization for CBIR

in hardware is far from trivial and may not always be the best design option.

As a matter of fact, such an optimization would be quite inefficient in practice, as the parallel distance evaluation can only proceed to the next candidate vector once all the distances score currently computed are discarded. The actual efficiency of the hardware would then decrease as the level of parallelism would grow, as depicted in Figure 2.7(b).

2.2.3. Mapping CBIR on the RDISK platform

We implemented the CBIR algorithm for the RDISK prototype in 2005. The critical point in the design of the filter was to optimize the workload balance in the execution pipeline represented in Figure 2.8. This pipeline involves the hardware filter and the soft-core processor, but its efficiency is also influenced by the disk I/O throughput (15 MB/s) and the network throughput (10 kB/s).

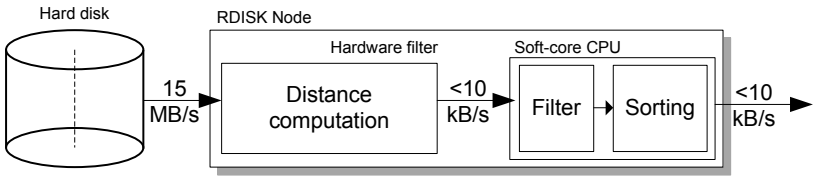


Figure 2.8.: CBIR execution pipeline on the RDISK architecture

Although the details and motivations for all our design choices is out of the scope of this document, we enumerate some of the most salient characteristics of our distance computation accelerator, shown in Figure 2.9.

– The baseline architecture is a processor array, in which each processing element handles σ independent query vectors, in a interleaved way (this corresponds to a LSGP partitioned processor arrays as explained in Section 5.3). Hence, the array only needs a new database vector element once every σ cycles. We chose the parameter σ such that the filter bandwidth does not exceed hard-drive throughput to avoid I/O stalls. We also took advantage of the embedded memory blocks available in the FPGA to store the query vectors and their identifiers.

– Because the soft-core processor was not able to sustain the distance computation stage, and because implementing a custom sorting within the FPGA would have required too much resource, we made the choice of taking advantage of threshold optimization described earlier, even though its efficiency is limited.

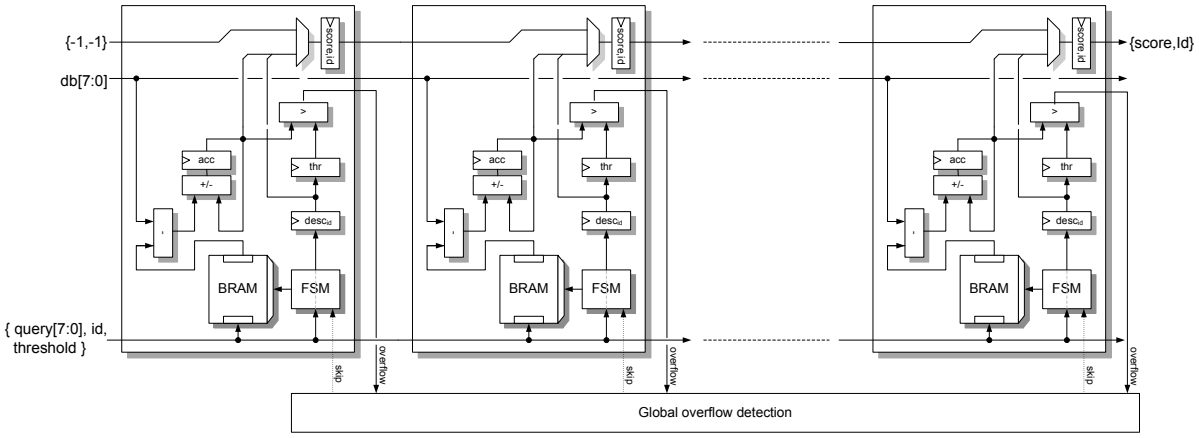


Figure 2.9.: Distance computation block for the RDISK architecture

We have been able to implement a 36 processors array in the FPGA, in spite of the relatively limited resource available. The speedup over the software implementation is x5 for a single RDISK node, leading to a global performance improvement between x100 and x200 for a full 48 RDISK cluster, depending on the number of descriptors in the query image.

2.2.4. Mapping CBIR on the ReMIX platform

The second implementation targeted the ReMIX prototype where the architectural constraints were completely different. In particular, due the high bandwidth at the Flash memory output (640 MB/s), there was no need to use LSGP partitioning (see Section 5.3) to reduce the architecture throughput as for RDISK. Additionally, the relative abundance of resource on the FPGA allowed us to consider a massively parallel architecture, consisting of multiple pipelines within a single FPGA, as illustrated in Figure 2.10.

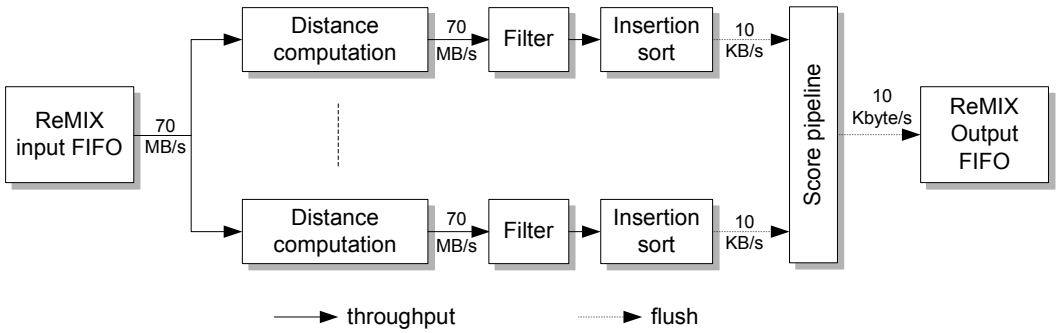


Figure 2.10.: System level view of the whole ReMIX CBIR search pipeline

The main difference, as shown in Figure 2.11, is that the ReMIX distance accelerator does not leverage dynamic threshold as in the case of RDISK. Our motivation for disabling this optimization was the area overhead induced by the handling of the threshold. This overhead had in turn a negative impact of the number pipelines that could fit on the FPGA, and we

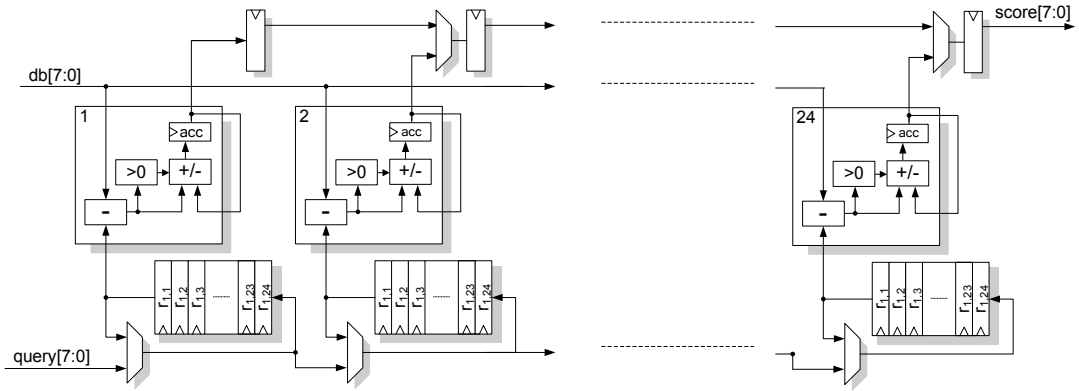


Figure 2.11.: Distance computation block for the ReMIX architecture

observed that the (limited) performance benefits of the optimization did not compensate for the loss of parallelism. In our distance computation processor, a processing element outputs a new score every 24 cycles. Therefore at most 24 processors can be implemented within a processor array if we want to avoid I/O conflicts on the score output pipeline.

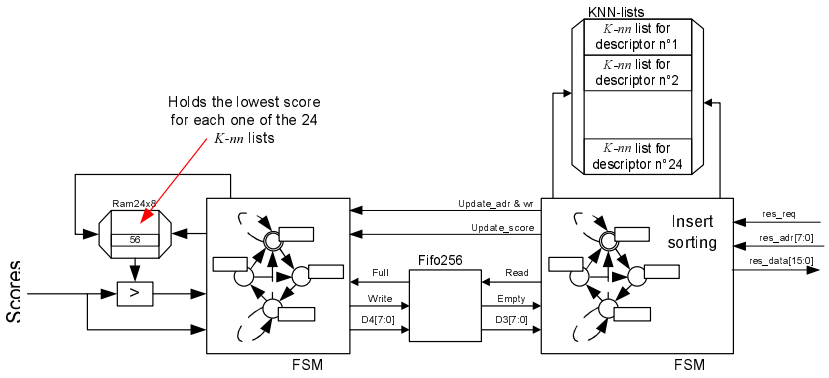


Figure 2.12.: Filtering and sorting blocks for the ReMIX architecture

This continuous stream of results also poses additional challenges to the filtering and sorting steps, which now also need to be performed on-the-fly. For this purpose, we have designed a custom component described in Figure 2.12. This component holds (and updates) a list of 24 thresholds to filter out the results of the distance component. It is also in charge of updating the sorted list of nearest neighbors for each of the 24 descriptors handled by the processor array. Thanks to the highly selective nature of the filtering process, this sequential sorting component is able to sustain the average throughput from the distance computation component, and does not cause any stall in the pipeline.

Bitwidth	24 bits	16 bits	12 bits	8 bits	3 bits
Query Time (sec)	50	40	28.5	20	12.5
Speed-up factor	18	22	31	45	72
Number of pipelines	4	5	7	10	16

Table 2.1.: Search time and speed-up for varying bitwidth

The observed speed-up factor over the original software implementation (with the early exit

optimization) was 45. In other words a single ReMIX system is as efficient as a cluster of 45 PCs. While this acceleration factor only holds for descriptors encoded as 8 bit integers, we estimated the corresponding results for different bitwidth by implementing as many processor lines as possible on the FPGA and use this result to estimate the corresponding speed-up. These performance projections are summarized in Table 2.1.

2.3. Discussion

In this section, we briefly discuss, in retrospect, the contributions and results presented in this chapter, starting by our work on reconfigurable platforms.

Discussing the RDISK and ReMIX projects six years after their end turned out to be instructive and a bit disappointing at the same time. One observation is that, as we speculated, Flash memory would replace magnetic drives sooner or later. While this prediction is not completely realized, it is definitely in its way, as the SSD market is literally booming. Still, only few attempts have actually been made to fully take advantage of the opportunities they offer as very high-performance custom mass storage, by fully utilizing the aggregate bandwidth of several chips, rather than staying tied to a standard hard-disk interface such as SATA. A notable exception is the Fusion-IO company [6], which provides high performance PCI-express solutions based on flash memory.

Another observation is that, in spite of all its potential benefits, the idea of coupling processing power to storage device (and to use FPGA for database processing) is still not mainstream. Although FPGAs are commonly used in high performance storage systems, their role is still restricted to low-layer error correction and/or arbitration. However, the idea of attaching a FPGA to a hard disk drive for coprocessing was successfully demonstrated by the Netezza company [33] in 2006 (now owned by IBM), with their SPU blade, which is strikingly similar to our RDISK prototype.

Nevertheless the impact of our work remained limited, and seems to have been ignored (in spite of its obvious relevance and similarity) by almost all recent contributions advocating the use of reconfigurable technologies for database processing platforms [141, 101, 49, 113]. This is however not so surprising and actually shed the light on a questionable habit in the custom computing community to often too quickly claiming for (re)inventing the wheel.

The work that we carried in these two projects required a huge investment in development effort, and one may conclude that designing and proposing original and innovative custom hardware architectures is not a viable research topic. Such as statement however need to be nuanced. This remains an important research topic, which currently suffers from one big issue: building real hardware prototypes for such systems is now out of reach for academic research. In the mean time, providing quantitative evidence on a prototype to backup the relevance of a new type of machine is mandatory. In my opinion, rather than trying to build real machines, custom computer architects should follow the way processor architects have been doing for more than a decade, and what supercomputer designers now also do: basing their experimentation on accurate simulators and/or models. These simulations could be carried at several different levels of abstraction of the machine (for example VPR for modeling the FPGA circuit [106], SocLib [155] for gathering cycle accurate platform level performance metrics, Orion [160] for modeling complex interconnect, etc).

Regarding our work on CBIR, it turns out to be of little practical interest today, as there is no more need for hardware accelerators in CBIR systems. Over the last few years, the *curse of dimensionality* was partially warded off and there now exist methods that reduce search time

6. http://www.fusionio.com

by orders of magnitude (to a few ten of milliseconds) while enabling very accurate results [99]. As a matter of fact, the only place where the use of FPGAs could still make a lot of sense is in the extraction of such descriptors, that requires a lot of processing power and relies on well known image processing algorithms that would be a good match for hardware acceleration [81].

Summary	
	Supervision: Auguste Noumsi, graduated in 2010, 20 %, now assistant professor at University of Douala, Cameroon
	Journal: Parallel Computing 2006 [20]
	International Journal of Electronics 2007 [29]
	Conferences: Engineering of Reconfigurable Systems and Algorithms 2003 [21]
	26th IEEE International Parallel & Distributed Processing Symposium 2006 [23]
	International Workshop on Applied Reconfigurable Computing 2007 [3]

Chapter 3

Massively Parallel Accelerator for HMM Profile Base Sequence search

The following chapter discusses contributions on the design of hardware accelerators for Hidden Markov Models (HMM) profile based sequence search, a widely used application in bioinformatics. The work presented below started in late 2006 in the context of the ReMIX and RDISK projects, while we were looking for new bioinformatics target applications. HMM based profile search, and more particularly the HMMER suite, turned out to be a challenging problem in terms of parallelization. I started working on the problem on my own (with support from Patrice Quinton), and the results lead to an initial contribution [9, 10] presented in Section 3.2. The results were later extended by Naeem Abbas (supervised by Patrice Quinton) in the context of his PhD [1]. This later work was carried in close collaboration with Sanjay Rajopadhye from Colorado State University.

This chapter is organized as follows, we will first start with a short presentation of the application (namely HMM Profile Bases similarity search), by explaining its role in bioinformatics and by detailing one of its most compute intensive kernel. In Section 3.2, we show how it is possible to derive an efficient processor array architecture for this application, even though the kernel does not admit any parallel schedule. The following section focuses on the kernel itself, and proposes a rewriting of the algorithm to make it amenable to an efficient parallel execution scheme based on *prefix-scan* operations. The chapter is closed by a survey of contributions related to ours, followed by a discussion on the relevance of FPGA based hardware accelerators in high performance biocomputing infrastructures.

3.1. Profile based sequence comparison

Sequence homology search tools are one of the most important kind of applications in computational molecular biology. In such applications, the protein sequences of unknown characteristics are compared against database of known sequences in order to predict protein functions and to classify protein families.

For this problem, techniques based on profile Hidden Markov Models (HMMs) [127, 62, 93] have shown to give very good results. Their strength stems from the fact that multiple sequence alignments concentrate on the features or key residues conserved by the family of sequences. As a consequence a profile based search can find a remote sequence homology that could not be detected through a pairwise alignment using simpler sequence matching algorithms such as BLAST [36] and/or Smith & Waterman [140].

A profile HMM is a regular HMM consisting of a sequence of columns. In each column, a *match* state models the allowed residue, an *insert* state models the insertion of one or more residues, and *delete* the deletion of a residue. An example of such a profile is given in Figure 3.1, in which bold arrows represent a possible path for the target Amino acid sequence (`CTTACGCTA`). Each path leads to a similarity score, the goal being the retrieval of the path with the highest score using a modified Viterbi algorithm.

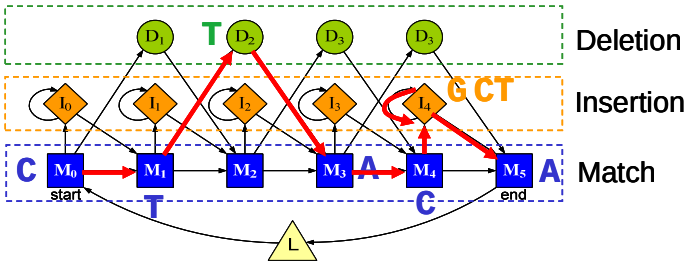


Figure 3.1.: Structure of a Plan7 profile HMM, as used in HMMER.

3.1.1. The HMMER software suite

One of the most commonly used program for profile based sequence analysis is the open source software suite HMMER, developed at Washington University, St. Louis by Sean Eddy [61]. Two versions of the tool are currently used by the community, HMMER 2.0 and a recent (March 2010) HMMER 3.0.

In HMMER 2.0, performance profiling shows that the program spends more than 97 % of the execution time in the `P7Viterbi` kernel. This kernel computes a similarity score between the profile HMM and the sequence at hand using a Viterbi dynamic programming algorithm. Matching a single HMM against a protein database is a very time consuming process, which is repeated many times during intensive comparisons.

HMMER 3.0 is a complete redesign of the program and of its algorithms. Among other optimizations, it introduces a new heuristic routine called `MSV`, that is used as a pre-filtering step to the more costly `P7Viterbi` algorithm. This new execution pipeline is described in Figure 3.2. It is important to understand that this new version (and their corresponding algorithms) has been carefully redesigned to benefit from an efficient implementation on recent x86 multi-cores (this is discussed in more detail in Subsection 3.3.3).

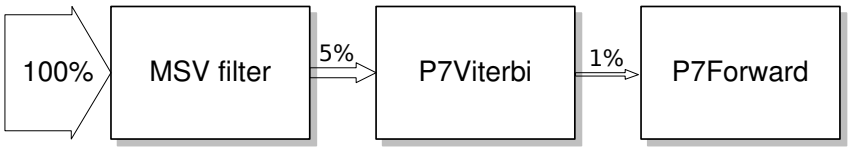


Figure 3.2.: The HMMER3.0 pipeline, showing the effect of the `MSV` filter, with %95 of the sequences being filtered out by the heuristic.

Performance figure in sequence comparison algorithms are generally measured in Giga Cells Updates per Second (GCUP/s), where a cell update corresponds to one update of the equations of the algorithm. Table 3.1 shows performance figures for HMMER 2.0 and HMMER 3.0 with and without the use of Intel SSE - SIMD extensions. The reader may find that the impact of SIMD extension on performance to be striking, as it alone brings more than ten fold improvement over the non-SIMD version.

HMMER	globin.hmm (M=143),	Pkinase.hmm (M=255),	rrm.hmm (M=77),	fn3.hmm (M=84)
V2	≈ 0.03	≈ 0.03	≈ 0.03	≈ 0.03
V3-noSSE	0.3	0.37	0.3	0.26
V3-SSE	5.2	7.16	2.83	2.65

Table 3.1.: Performance in GCUPS for Pfam-B.fasta

$$M_{i,k} = \max \left\{ \begin{array}{l} e_M(\text{seq}_{i,k}) + \max \left\{ \begin{array}{ll} M_{i-1,k-1} & +\text{TMM}_k \\ I_{i-1,k-1} & +\text{TIM}_k \\ D_{i-1,k-1} & +\text{TDM}_k \\ B_{i-1} & +\text{trB}_k \end{array} \right. \\ -\infty \end{array} \right. \quad (3.1)$$

$$I_{i,k} = \max \left\{ \begin{array}{l} e_I(\text{seq}_{i,k}) + \max \left\{ \begin{array}{ll} M_{i-1,k} & +\text{TMI}_k \\ I_{i-1,k} & +\text{TII}_k \end{array} \right. \\ -\infty \end{array} \right. \quad (3.2)$$

$$D_{i,k} = \max \left\{ \begin{array}{l} M_{i,k-1} + \text{TMD}_k \\ D_{i,k-1} + \text{TDD}_k \\ -\infty \end{array} \right. \quad (3.3) \qquad E_i = \max \left\{ \begin{array}{l} max_{k\in[0,K]}(M_{i,k}) + \text{trM}_k \\ -\infty \end{array} \right. \quad (3.4)$$

$$N_i = \max \left\{ \begin{array}{l} N_{i-1} + \text{trCnst}_0[N] \\ -\infty \end{array} \right. \quad (3.5) \qquad J_i = \max \left\{ \begin{array}{l} E_i + \text{trCnst}_0[E] \\ J_{i-1} + \text{trCnst}_0[J] \\ -\infty \end{array} \right. \quad (3.6)$$

$$B_i = \max \left\{ \begin{array}{l} N_i + \text{trCnst}_1[N] \\ J_i + \text{trCnst}_1[J] \\ -\infty \end{array} \right. \quad (3.7) \qquad C_i = \max \left\{ \begin{array}{l} C_{i-1} + \text{trCnst}_0[C] \\ E_i + \text{trCnst}_1[E] \\ -\infty \end{array} \right. \quad (3.8)$$

Figure 3.3.: The system of recurrence equations defining the `P7Viterbi`kernel

Figure 3.3 shows the system of recurrence equations defining the dynamic programming algorithm used for comparing a sequence against a profile (known as `P7Viterbi` kernel), where L denotes the protein sequence length and M the length of the profile HMM. The key observations of the equations (3.1-3.8) are that

- there is a chain of dependences in the increasing order of k in computing the values of D in any column;
- to compute the E for any column, we need *all* the values of M of that column, each of which needs a D from the previous column; and
- the value of E of a column is needed to compute *any* M in the next column.

Because of this, there seems to be an inherent sequentiality to the algorithm, preventing its parallelization. The same holds for the `MSV`routine, even though the use of reduction operations [1] is a simple way for exposing more parallelism. The remainder of this chapter describes two contributions, which target the same problem: *How to accelerate the HMMER application in spite of its apparent lack of parallelism* ?

3.2. Design space exploration for parallel HMMER

In this section, we will show how to use well known results from the automatic parallelization community, to derive an efficient parallel architecture for the `P7Viterbi`kernel. This derivation is done incrementally, by successive refinements of the initial parallelization.

3.2.1. Exploring scheduling/mappings

Deriving a regular parallel architecture from a System of Affine Recurrent Equation (SARE) is a well studied problem, for which many techniques have been proposed. The derivation

1. A reduction corresponds to a min/max/sum operation on all elements of a collection.

generally uses affine (or quasi affine) space-time mappings. Space-time mappings associate to every indexed variable of the SARE:

- A *logical execution* time instant that we call *schedule*. In the scope of this work, we target one dimensional schedules, that take the form of an affine function of the indices written as
$$s(i_0, \ldots, i_m) = s_0 i_0 + \ldots + s_m i_m.$$
- A *physical location*, i.e, coordinates in a processor space. This location is also expressed in the form of an affine function of the indices (that we call *allocation function*). In our case, we are only interested in one-dimensional (i.e linear) arrays. We therefore have
$$p(i_0, \ldots, i_m) = \alpha_0 i_0 + \ldots + \alpha_m i_m.$$

Of course, this space-time mapping must satisfy several conditions.

- The chosen schedule must enforce all data dependencies present in the SARE. For $u, v \in \mathcal{D}$ with v depending on u, the schedule function must guarantee that $s(v) > s(u)$.
- The space-time mapping must be conflict-free, i.e, there must be no u and v in $\mathcal{D}$ such that $v \neq u$, $s(u) = s(v)$ and $p(u) = p(v)$.
- The space-time mapping must be *dense*, which means that processors on the array are active every cycle in steady state.

It can be easily shown that the best (i.e, the fastest) schedule [2] which can be found is $s(i, k) = Mi + k$, which corresponds to the original algorithm sequential schedule [126]. It has little practical interest, apart from convincing us that there is, at first glance, no parallelism in this algorithm.

Although loop level parallel schedules cannot be found in the `P7Viterbi` routine, we observe that running the `hmmsearch` tool consists in completely independent matchings of a single HMM against a large number of input sequences: these matchings can therefore run in parallel. We model this additional parallelism by adding a new index to the SARE that identifies instances of the kernel running in parallel. This transformation is illustrated in Figure 3.4.

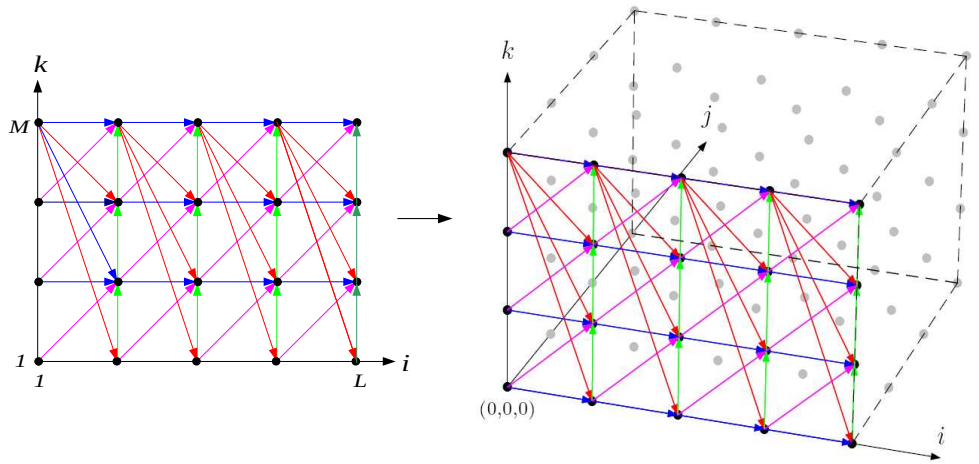


Figure 3.4.: Adding a dimension to the schedule

We have taken advantage of this property to explore various scheduling/mapping strategies. More precisely, we have obtained our final schedule through successive refinements of the space-

2. For the sake of readability, the constant M is left as a symbolic constant, in practice, M must be a compile-time constant.

Design	#PE	Memory per PE (in bytes)	#pipeline stages
Wave front ($\sigma = 1, \lambda = 1$)	N	$O(MA)$	1
Interlaced ($\sigma = 1, \lambda = 1$)	M	$O(M + A)$	1
Partitioned ($\lambda = 1$)	$\lceil M/\sigma \rceil$	$O(M + \sigma A))$	1
Pipelined	$\lceil M/\sigma \rceil$	$O(\lambda M + \lambda \sigma A)$	λ

Table 3.2.: Summary of the proposed Space-Time Mappings

time mappings. A graphical representation of these schedules is given in Figure 3.5. Detailing these mappings is out of the scope of this document, we will limit ourselves to a short description of their characteristics.

The first schedule in Figure 3.5.(a) is a *Wavefront schedule*, the most straightforward parallel schedule that can be derived. It simply consists of running several independent instances of the algorithm on the hardware. This approach suffers from conflicting memory accesses to the transition score table, that impose these tables to be duplicated in every processing element. Given the large size of these tables, this approach is impractical, and we therefore explored alternative schedules to avoid such conflicts.

This was achieved in our second schedule, which we call an *Interlaced schedule*. In this schedule, shown in Figure 3.5.(b), a given processor only accesses a small fraction of the transition table, enabling the distribution of this table among all the processors in the array. This approach forces a mapping in which the number of processors is equal to the size of the profile at hand. Given that the average profile size is 200 (see Figure 3.7(a)), this again makes the approach impractical.

We hence applied a LSGP (Locally Sequential Globally Parallel) partitioning transformation to address the issue. LSGP consists in tiling the iteration domain to merge several processors of the *virtual array* obtained by the initial scheduling, into clusters of *physical arrays*, as illustrated in Figure 3.5.(c). This transformation allows designers to control the amount of parallelism in the application simply by choosing the cluster size (that we refer to as σ).

The last transformation is shown in Figure 3.5.(d); its role is to transform the partitioned schedule to increase the delay (in cycles) between two dependant iterations in a given processor. This extra delay is then used to pipeline the processor datapath to increase the array achievable clock frequency.

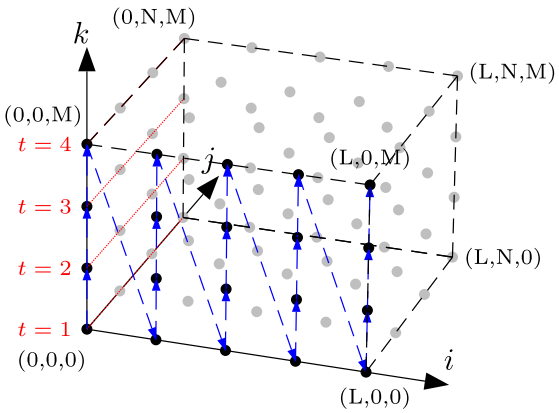
A summary of the characteristics for the proposed space-time mappings is given in Table 3.2. In this table, M is the length of the profile HMM, N is the number of instances, A is the number of amino-acids ($A = 25$), σ is the size of the cluster, and λ is the number of pipeline stages within each processor. The resulting processor array architecture, corresponding to the schedule of Figure 3.5.(d), is given in Figure 3.6.

3.2.2. Experimental validation

Previous subsection has shown that the hardware resource usage (in terms of logic cells and memory) of our hardware accelerator is highly dependent on the size M of the HMM model at hand, and on design parameters such as the pipeline level λ, or the partitioning factor σ. In order to have a more quantitative view of this resource cost, we implemented several configurations of the processor array on a Xilinx Spartan3-4000. We tried, for each implementation, to maximize the size of the processor array given some value of M and λ, for a chosen wordlength of 15 bits. The results, shown in Table 3.3, indicate that for larger values of M, the limiting factor is the number of embedded memory blocks available on the target FPGA device (BRAM column).

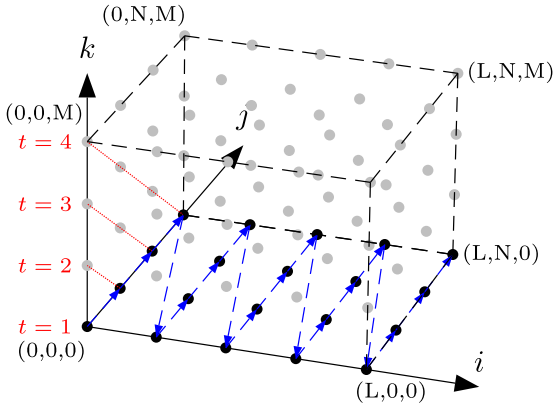
In our implementation, each PE performs one iteration per cycle, with clock frequencies ranging from 40 MHz to 60 MHz. This has to be compared to the reported software performance

(a) Wavefront schedule



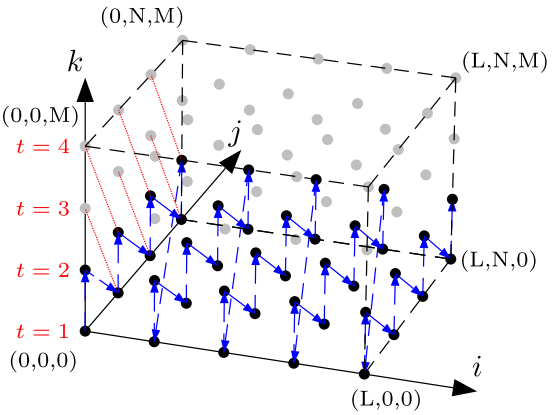
$$s(i,j,k) = Mi + k \ ,$$
$$p(i,j,k) = j \ ,$$

(b) Interleaved schedule



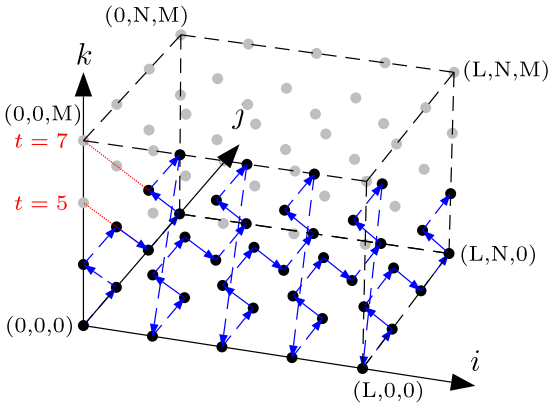
$$s(i,j,k) = Mi + j + k \ ,$$
$$p(i,j,k) = k \ .$$

(c) Partitioned schedule



$$s(i,j,v,p) = \sigma M' i + \sigma j + v + \sigma p \ ,$$
$$p(i,j,v,p) = p \ .$$

(d) Partitioned & Pipelined schedule



$$s(i,jj,l,v,p) = \sigma\lambda M' i + \sigma(\lambda jj + l) + \lambda v + \lambda\sigma p \ ,$$
$$p(i,j,v,p) = k \ .$$

Figure 3.5.: Space-time mappings for the `P7Viterbi` algorithm. The dashed blue lines represent the schedule (i.e the order of visit) of iterations within the domain, whereas the dotted lines correspond to iterations executed in parallel

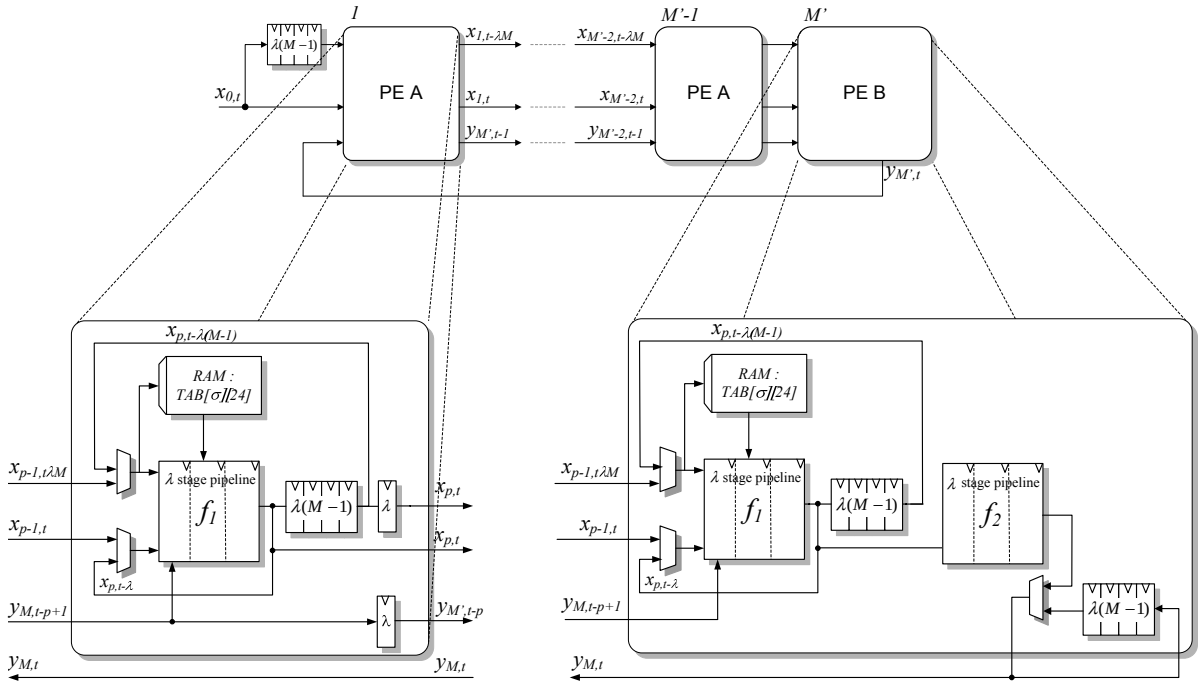


Figure 3.6.: Architectural template for the HMMER processor array. The shift registers depths and the number of processor are expressed as functions of σ, λ and M

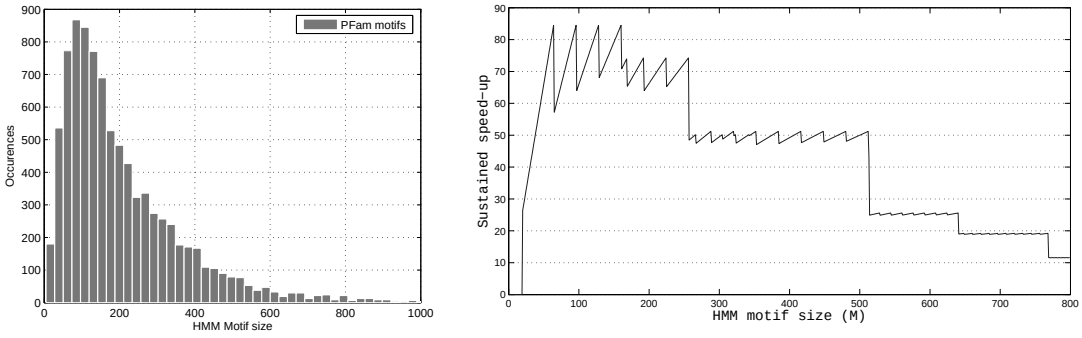
M	σ	N_{PE}	λ	BRAM	Slices	f_{max}	GCUP/s	speedup
60	2	30	3	60 / 63%	16453 / 59%	60	1.8	71
120	4	30	3	90 / 93%	16534 / 59%	55	1.7	68
190	6	32	2	96 / 100%	18640 / 67%	55	1.8	70
250	8	32	2	96 / 100%	18127 / 65%	55	1.8	70
380	11	32	1	96 / 100%	20682 / 74%	40	1.28	51
500	16	32	1	96 / 100%	20171 / 72%	40	1.28	51
600	38	16	1	96 / 100%	12844 / 46%	40	0.64	25

Table 3.3.: Resource usage and performance for various motif sizes

of HMMER V2.0 back in 2006, which was 24 MCUP/s on an Intel P4 CPU [117]. The last two columns of Table 3.3 give peak performance estimates for our accelerator (we assume that the hardware is clocked at its maximum clock frequency) and corresponding speedup over the software implementation.

However, these performance gains should be balanced by the fact that, although it is possible to use a HMM profile of size M with an architecture that accommodates a maximum model size of M_{max}, only a fraction M/M_{max} of the computations actually contributes to the matching score. For example, when $M = 50$ and $M_{max} = 64$, the sustained performance slips to only 78% of the peak performance.

This situation favors the use of reconfigurability: for a given value of M, we can choose among a library of preexisting bitstreams the configuration which offers optimal performance for that very specific value of M. Of course, such a strategy induces a hardware reconfiguration overhead. However, given current reconfiguration latencies (125 ms for a Spartan3-1000) its impact on overall performance is very limited. On the other hand, the fact that M_{max} is constrained by $M_{max} = \sigma N_{PE}$ strongly restricts the number of possible configurations, and often prevents to find a perfect match for M_{max}. To illustrate this side effect, we estimated



(a) HMM model size distribution

(b) Speedup as a function of M

Figure 3.7.: Impact of profile model size on the resulting architecture

the optimal sustained performance that could be achieved by our accelerator for values of M in the range $[50, 640]$, for a clock speed of $33MHz$. The results, given in Fig. 3.7(b), indicate that sustained performance is significantly impacted by this phenomenon, especially for smaller values of M.

3.3. Parallelizing through max-prefix inference

The approach that we presented in previous section relies on the fact that in practice, several independent instances of the `P7Viterbi` kernels can be executed in parallel, providing additional parallelism. This poses several problems in the case of large arrays with deeply pipelined processors. For a 150 processor array with 3-stage pipelined processors, we have to handle 450 `P7Viterbi` instances. This results in huge memory costs, leading to a sub-optimal use of FPGA computational resource.

In this work, done in collaboration with Sanjay Rajopadhye in the context of the PhD of Naeem Abbas (supervised by Patrice Quinton), we propose a technique to rewrite the kernel in such a way that it becomes amenable to parallel implementation at a price of a moderate, constant factor increase in the computational volume. We then propose several strategies for efficiently implementing this algorithm on an FPGA-based High Performance Computing platform using High-Level Synthesis tools and discuss the performance that we obtained.

3.3.1. Finding hidden parallelism in `P7Viterbi` kernel

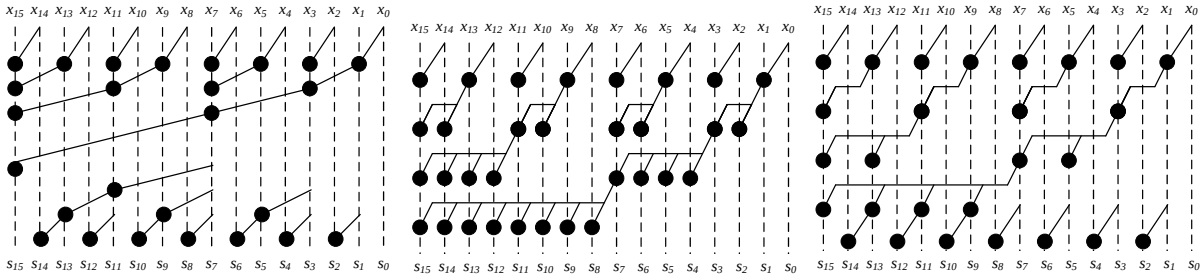
Rather than focusing on the dependency between X and M (related to the feed-back loop), we have developed an alternate formulation of the equations enabling a scalable parallelization. For our purposes, we shall focus on the equation of $D_{i,j}$, defined as follows:

$$D_{i,k} = \begin{cases} k = 1 : M_{i-1}[0] + A'[0] \\ k > 1 : \max(D_i[k-1] + B[k], \\ \qquad\qquad M_{i-1,k} + A'[k]) \end{cases} \tag{3.9}$$

We have shown that, using algebraic rewriting [1], it is possible to replace the chain of dependency of $D_{i,k}$ on $D_{i,k-1}$ by an expression involving a *max-prefix* operation. Prefix computation is a general class of computations formally defined as follows. Given an input vector x_i with $0 \leq i < N$ we define its $\oplus$-prefix vector y_i as

$$y_i = \bigoplus_{k=0}^{i} x_k = x_0 \oplus x_1 \oplus \ldots \oplus x_i,$$

where $\oplus$ is a binary operator with associativity (and possibly commutativity, see [91] for a more detailed definition). Prefix computations are very interesting in our context, as they can be easily parallelized. Parallel prefix networks is a well studied problem, and there is a wealth of research dealing with fast (i.e, parallel) implementations of prefix adders [96, 47, 92, 76, 139] some of them being illustrated in Figure 3.8. For a comprehensive classification, describing the trade-offs in existing network topologies, we invite the reader to refer to the work of Harris [79].



(a) Brent-Kung (b) Sklansky (c) Ladner-Fischer

Figure 3.8.: Examples of parallel prefix implementation for $N = 16$

One of the most important aspects of these network topologies is that they allow the designer to explore the trade-off between speed (i.e. critical path of the resulting circuit), area (number of operators used to implement the prefix operation), and other metrics such as fan-out or wiring length. For example, Figure 3.8.(a) shows a Brent-Kung [47] network that computes the prefix in $2(\log_2 N - 1)$ stages with $2(N-1) - \log_2 N$ operators. Similarly, Figure 3.8.(b) shows a Sklansky network which implements a faster circuit ($\log_2 N$ stages) at a price of an increase in area ($\frac{N}{2} \log_2 N$ operators).

To help the reader in understanding the benefits of this rewriting transformation, we provide an illustration of the data dependence flow in the rewritten algorithm for a small problem size (profile size $N = 8$) in Figure 3.9 . The reader can observe that the dependency $D_{i,k} \longrightarrow D_{i,k-1}$ in equation (3.3) is now converted to a *max-prefix* block, reducing the critical path from $O(N)$ to $O(\log_2 N)$ operations. Another consequence is that update operations for $M_{i,k}$, $I_{i,k}$ and $D_{i,k}$ can be executed in parallel for all values of k in the domain $0 \leq k \leq N$.

3.3.2. Mapping the full HMMER 3.0 pipeline to hardware

It can be easily seen from Figure 3.9 that in `P7Viterbi`, it is not possible to pipeline the execution of consecutive stages —all the results of the i^{th} stage are needed before *any* value in the (i+1)th stage can be computed. Despite the fact we replaced the initial chain of dependency of $O(M)$ operations by a chain of $O(\log_2(M))$, the possibly large values of M may impact clock frequency. Similarly, because of obvious resource constraints, it is not possible to directly map the whole data-flow graph of Figure 3.9 on the FPGA.

We have used two circuit transformations to address these issues, namely *C-Slow* and *Tiling*. The *Tiling* transformation is closely related to the partitioning of processor arrays presented in Section 3.2. Similarly, *C-Slow* enables the overlapped execution of multiple kernel instances. A summary (following the spirit of 3.2) of the impact of these transformations on the architecture resource cost and performance is given in Table 3.4

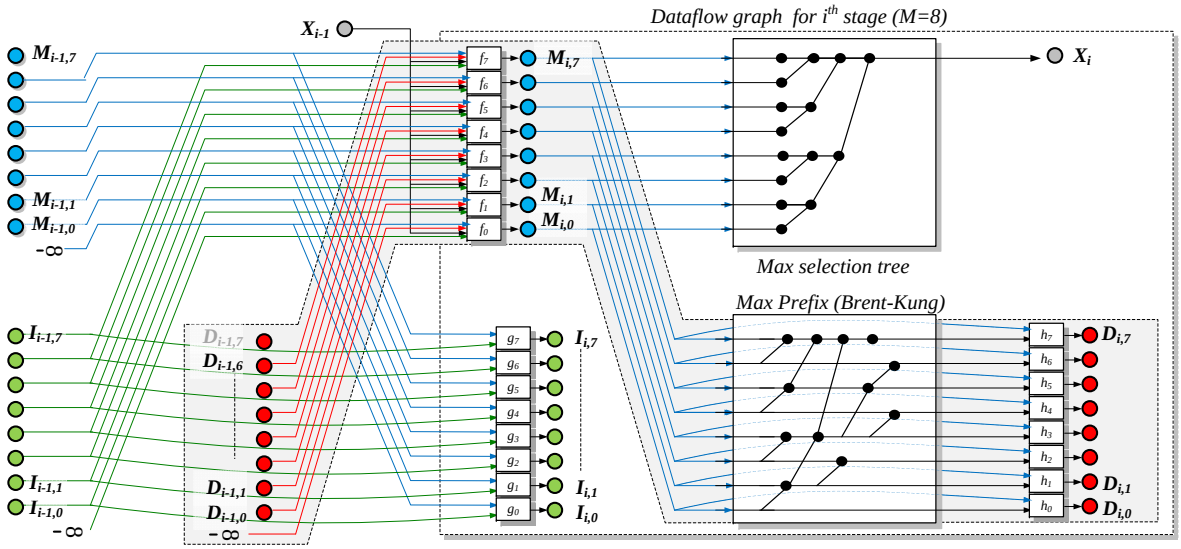


Figure 3.9.: Dataflow graph for one stage of the kernel ($N = 8$) after rewriting.

Method	Area	T_{clk}	throughput
Combinational	$O(M)$	$O(\log_2 M)$	$O(\frac{M}{\log_2 M})$
Tiled	$O(M/P)$	$O(\log_2 \frac{M}{P})$	$O(\frac{M}{\log_2 \frac{M}{P}})$
C-slow	$O(M)$	$O(1)$	$O(M)$
Tiled + C-slow	$O(M/P)$	$O(1)$	$O(M/P)$

Table 3.4.: Summary of the different architectures and their space-time characteristics

3.3.3. Experimental results

Our target execution platform consists of a high-end FPGA accelerator from XtremeData (XD2000i-FSBFPGA) with two Stratix-III FPGAs. The circuit was entirely designed using a commercial C to Hardware compiler (Impulse CoDeveloper C-to-FPGA). The use of a High-Level Language as input specification helped us exploring large portions architectural design space in a short amount of time, while staying very competitive in terms of performance and resource usage compared to a HDL implementation.

A system level view of the full HMMER3 pipeline is given in Figure 3.10. It contains five blocks, three `MSV` blocks communicating with two `P7Viterbi` blocks). In this implementation, the role of `Sequence MUX/DEMUX` is to merge/distribute interleaved sequences among computation blocks and the role of the `Filter` blocks is to filter out all sequences with scores lower than the threshold. Performance and area given in Table 3.5 show that speedups of $\times 7.5$ over a 3 GHz dual-core can be achieved on a single [3] FPGA.

M	P	MSV	P7Vit	Logic Util.	M9K	MLAB	MHz	GCUPS
64	8	6	1	128K / 63%	864 / 100%	215Kb	103	40.0
64	8	7	1	143K / 70%	864 / 100%	269Kb	97	44.2
128	8	3	1	105K / 52%	864 / 100%	181Kb	95	37.2
256	8	2	1	147K / 72%	864 / 100%	203Kb	99	51.5
512	8	1	1	79K / 39%	675 / 78%	66Kb	89	45.7

Table 3.5.: Performance and area for our system-level implementation

3. It turns out that the board does not support the combined use of the two FPGAs

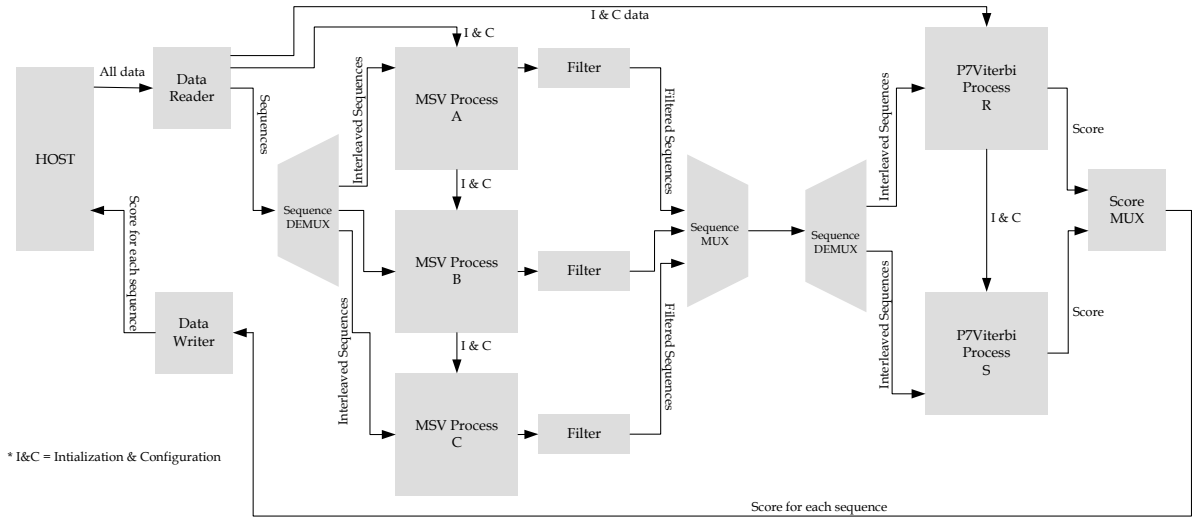


Figure 3.10.: System level view of a complete HMMER3 hardware accelerator

This modest speedup can be explained by the fact that the new SSE implementation of HMMER is highly optimized. An important point to make is that HMMER 3.0 was not simply ported to take advantage of SIMD extensions. As a matter of fact, there also exists such a port for V2.0 that brings only marginal performance improvement. In the 3.0 version, the entire algorithm, and in particular the data encoding, were completely re-engineered to perfectly match the Intel SIMD extensions.

Another important point of discussion is how the performance of our FPGA implementation would perform compared to an equivalent GPU implementation. This is an important question, since GPU offer more flexibility at a much lower cost than a typical HPC FPGA platform, albeit at a higher cost in power/energy consumption. Unfortunately, there is currently no GPU version of HMMER3 available for comparing the two targets.

However, we believe that, contrary to HMMER2, a GPU version of HMMER3 would only offer marginal speedup w.r.t. the optimized SSE version. The GPU speedup for HMMER2 were reported to be in the order of $\times 15 - \times 35$ [159] and $\times 20 - \times 70$ [70] for a single GPU over the software implementation. When looking at HMMER3 speedup results (given in Table 3.1), it turns out that the use of an optimized SSE software implementation alone brings a speedup factor of 20 over the non SSE version, thanks to the systematic use of sub-word parallelism. This is comparable to the reported speedup of GPUs implementation for HMMER2, and as GPUs do not have support for short integer sub-word parallelism, it is therefore unlikely that GPUs will do much better than the SSE implementation neither in term of performance nor in terms of energy efficiency.

3.4. Related Work

Because of its execution time, and of its wide use the bioinformatics community, there have been several attempts to accelerate the `P7Viterbi` kernel, either using SIMD extensions [104], parallel machines [158], GPUs [83, 159, 70], Network Processors [162] or FPGAs [107, 117, 118, 147, 146, 9, 1]. We are however not aware of any work (other than ours) on the whole HMMER 3.0 pipeline. The following section provides a short review of these approaches, with a focus on the way authors have dealt with the lack of apparent parallelism in the `P7Viterbi` kernel.

In order to simplify its parallelization, some authors [117, 107] have suggested modifying the original `P7Viterbi`algorithm by removing the outer loop carried dependency, which forbids the use of loop level parallelism. However, this *feedback-free* modified algorithm induces a lot of false-negative matches (some potentially interesting hits are *missed* by the algorithm). Oliver et al. were the first to propose an approach [118] to handle the `P7Viterbi` feedback loop in 2007, however, their architecture is not scalable as it requires a crossbar for enabling concurrent access of processors to transition cost tables.

In 2009, an approach based on speculative execution was proposed by Takagi et al. [147] and Sun et al. [146]. Their idea is to speculatively ignore the dependency related to the feedback loop, as it seldom contributes to the actual scores in the next column, and recompute all mispredicted results whenever a mispeculation is detected. The approach has shown to be very effective in practice, although its efficiency largely depends on the characteristics of the query profile.

In 2010, Eusse Giraldo et al. [65] proposed another approach for accelerating `P7Viterbi` on custom hardware. They use a simplified (without J state) Viterbi kernel as a filter and passes only sequences with significant scores to the original Viterbi kernel along with divergence algorithm [41] data. The divergence algorithm data reduces the number of cells that must be calculated with the original Viterbi kernel by providing limits of the alignment region. The alignment region defines where the alignment starts and ends. This approach yields an acceleration of 5.8 GCUPS (Giga Cell Updates Per Second). However, the use of simplified Viterbi as a filter may not detect multiple hit alignments, as in the case of the feed-back free approach.

Also in 2010, Ganesan et al. [70] proposed a GPU acceleration of `P7Viterbi` by breaking the chain of dependency inside the kernel, following an approach that turns out to be very similar to the one we presented in this chapter [4]. The reported speed up over a factor of 100 times on 4 Tesla C1060 GPUs in comparison with software implementation of HMMER2. However, in contrast to their work, we leverage the full power of parallel prefix networks that provide the freedom to explore the trade off between delay and area cost according to the architecture requirements.

3.5. Discussion

This work has led to three publications, two in international conferences (ASAP 2007 [9], FPT 2010 [1]) and one in an international Journal (JVLSI 2010 [10]). The article submitted in 2007 to ASAP received the conference best paper award.

One conclusion is that, even for short width integer arithmetics, FPGA accelerators may not always significantly outperform handcrafted vectorized code. This is an important (negative) result, yet it seems that the HPC custom computing community is still not ready to accept this fact. Indeed, speedup claims of two order of magnitudes are often reported in the litterature. However the performance of these FPGA implementations is only seldom compared against a truly optimized software version (often simply because such an optimized implementation does not exist) leading to artificial/unrealistic speedup. Interestingly, this phenomenon also holds for GPGPUs.

We believe that the real strength of FPGA accelerators is their power efficiency more than their performance ($\approx 65\,\mathrm{W}$ for a Intel Core Duo vs $20\,\mathrm{W}$ for an FPGA accelerator). For our application, the FPGA implementation represents an estimated energy efficiency improvement of a factor of 25. However given the current relatively low cost of energy, this benefit alone does not compensate for the high price of these accelerators (between \$2k and \$5k for a single FPGA

4. Our paper was submitted in June 2010, and their work was published in august 2010

board).

The other conclusion is that C to Hardware tools do not only help decreasing design time. They can also lead to more efficient designs, as this increase in productivity allows the designer to explore a wider design space, leading him/her to ultimately find an architectural solution that he/she would have missed otherwise [4]. Nevertheless, these tools still do not hide the complexity of hardware design, and should rather be considered as tools for micro-architectural synthesis from C, rather than true C to gate compilers. This, even more than their limited performance improvement, is the current weakness of FPGAs for high performance scientific.

In the same way software designers now benefit from refactoring tools, which help them maintaining and restructuring their code, we believe there is a need for *hardware refactoring* tools, that would help HLS users to refine their target micro-architecture through the use of source-to-source transformations, a topic that we address in chapter 5.

Summary	
	Supervision: Naeem Abbas, graduation expected in 2012, 80 %,
	Journal: Journal of VLSI Signal Processing Systems, 2010 [10]
	Conferences: Application Specific Architectures and Processors 2007 [9]
	International Conference on Field Programmable Technology 2010 [1]

Chapter 4

Ultra Low Power Wireless Sensor Networks

The research topic presented in this chapter originates from several earlier work: the work carried on low power implementations of processor arrays carried during my PhD, the work of Ludovic L'Hours [100] dealing with the synthesis of custom processors from C programs, and other research activities of the CAIRN group on Wireless Sensor Networks (WSN). The main idea behind this work was to see whether it would be possible to take advantage of a combination of hardware customization and power gating to improve the power efficiency of current WSN platform architectures. Most of this work is the result of the PhD of Adeel Pasha (graduated in December 2010).

The chapter is organized as follows, we start by a description of wireless sensor network platforms and discuss the design challenges they pose. We show that existing programmable solutions are still not energy efficient enough to deal with these challenges. We then describe our approach, based on the notion of hardware micro-tasks, and summarize the main features of its supporting design flow. This presentation is followed by a quantitative analysis of the energy improvements of our approach, demonstrating its relevance. We conclude by a discussion of the merits, limitations, and research perspectives that our approach opened.

4.1. WSN platforms design challenges

Wireless sensor network (WSN) is a promising technology with potential applications in many domains of daily-life, such as structural-health and environmental monitoring, medicine, military surveillance, robotic explorations etc. A WSN is composed of a large number of sensor nodes deployed inside a region of interest or very close to it.

WSN nodes are low-power embedded devices consisting of processing and storage components (a processor connected to a RAM and/or flash memory) combined with wireless communication capabilities (Radio transceiver) and some sensors/actuators. Designing a WSN node is challenging, as designers must deal with strong form factor and energy constraints. For example, WSN nodes may need to run autonomously for months using only very limited energy sources (between ten and a few hundred mAh for small form factor Lithium batteries), or by harvesting energy from their environment (power supply of at best 20 μW).

Current WSN nodes are based on micro-controllers such as MSP430 [150] with typical power dissipation of a few mW, which are still orders of magnitude too high for many candidate applications of WSN. To address this power inefficiency, many WSN-specific controller implementations have been proposed by the academic community. These controllers try to exploit WSN-specific *event-centric behavior* and/or rely on transistor or circuit level optimizations (asynchronous logic, deep sub-threshold voltage) to achieve lower energy per instruction scores. Table 4.1 summarizes the energy efficiency of state of the art approaches.

Although some of them show impressive energy efficiency improvements, they suffer from many drawbacks. For instance, sub-threshold logic is highly susceptible to temperature, noise and process variations. Also, asynchronous logic is difficult to integrate in conventional design-flows. In some cases, it is the operating frequency range that becomes unacceptably slow (e.g.,

Processor	Voltage (V)	Frequency (MHz)	Energy (pJ/inst)	Normalized Energy (pJ/inst)	Process
SNAP/LE [64]	0.60	23	75	52	180
Phoenix [136]	0.50	0.1	2.8	2.8	180
Charm [138]	1.03	8	96	23	130
BlueDot [128]	NA	8	26	NA	130
MSP-like core [95]	0.50	0.4	27	27	65

Table 4.1.: Actual and normalized energy-efficiency for various ultra low-power WSN-specific controllers.

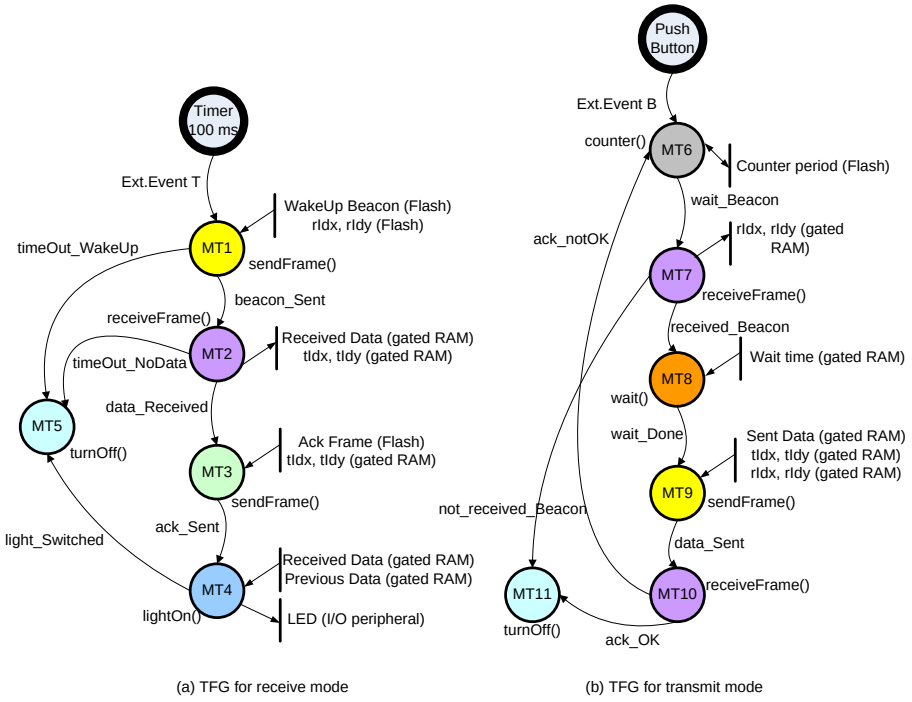


Figure 4.1.: TFGs presenting the micro-tasks running during a lamp switching application.

Phoenix processor can only operate at around 100 kHz). All of these approaches were manual circuit level (i.e. transistor level) implementations, hence limiting their practical usability. We believe that there is a need for an approach that relies on well understood circuit level optimizations (s.a power gating) and that is as much automated as possible.

To help the reader understanding our approach, we will use in the following a simple toy application (a remote lamp switching node). We assume that the application is modeled as a Tasks Flow Graph (TFG), where task execution is triggered by *events*, such events being external or produced by another task. We also restrict ourselves to tasks following a *run-to-completion* semantic, as in the case of TinyOS [119], a widely used Operating System in WSN. Figures 4.1.(a) and 4.1.(b) shows the TFGs of a lamp switching application (in receive and transmit mode), where a transmitting node demands a receiving node to switch on/off its lamp if a button is pressed at transmitter end. This application involves several tasks: data transmission, data reception, wait for acknowledgment, and timer, push button and lamp switching management, etc.

These control-oriented tasks involve sub-tasks that are spread across different layers of the OS/communication stack. For instance, beacon and data packets (transmission and reception) involve physical layer functions that exchange data, using a SPI-protocol, between the I/O

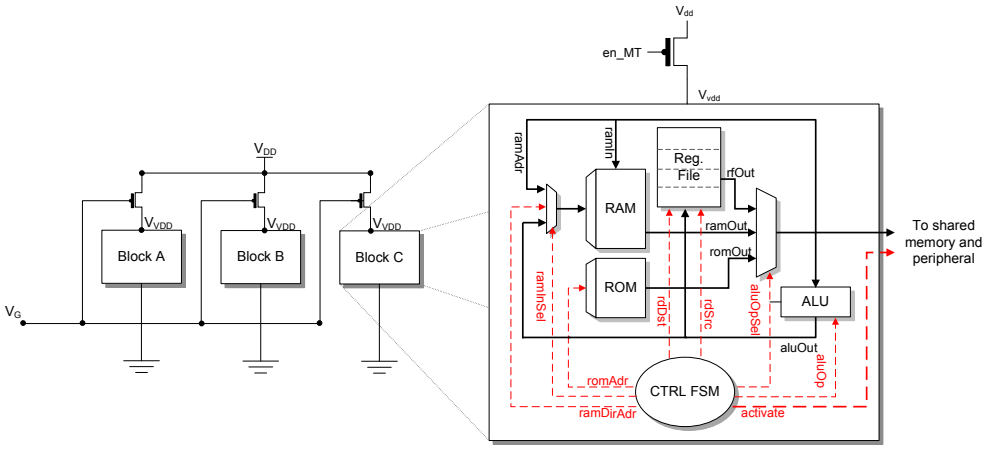


Figure 4.2.: Architecture of a power gated generic micro-task (here with an 8-bit data-path).

peripherals of the MCU and the Radio transceiver. The control flow itself follows a simplified version of RICER, a low-power MAC protocol [103].

4.2. A new hardware platform model for WSN

In this work, we propose to distribute the whole WSN node software framework into a set of *hardware micro-tasks* (possibly running concurrently), to benefit from parallelism while maintaining a high degree of specialization within each task. For example, a complete WSN communication stack uses approximately 3500 instructions on a MSP430. By distributing the stack functionality onto 7 micro-tasks, we can reach an average task size of 500 instructions, a granularity level at which we can expect energy improvements. This may come at the price of an increase in area and in static power dissipation, and we will show that our approach does not suffer from these side-effects.

In this approach a *micro-task* is implemented in the form of a custom micro-architecture synthesized from high-level behavioral (ANSI-C) specifications. The micro-architectural model used in our flow is illustrated in Figure 4.2, and some of its characteristics can be customized (ALU, datapath bitwidth, register file size, etc).

Although hardware customization is known to be a very efficient technique for reducing dynamic power, it was mainly used in the context of regular and compute intensive kernels. In this work, our interest rather goes to control dominated tasks (device drivers, MAC protocols, routing, etc.) which form the bulk of WSN workload.

To cope with the increase of static power induced by the use of multiple hardware tasks, we combine this customization with *power-gating*, a leakage power reduction technique, which consists in turning-off the power supply of idle components. This is realized by adding a *sleep transistor* between the actual V_{dd} (power supply) rail and the component's V_{dd} as illustrated in Figure 4.2. Needless to say, *Power-gating* is a valuable power reduction technique for WSNs, which do exhibit long idle periods (with duty cycles often below 1 %).

Our approach implies that all the micro-tasks are hard-wired into silicon as custom logic blocks. This makes post-production upgrade or bug fixing costly, Although flexibility is often of a great concern for WSN system designers. However, when looking more carefully at actual design practices, we can observe that the need for flexibility and reprogrammability is essentially geared towards the user application layer, which happens to represent only a small fraction of the WSN node processing workload, this workload being almost entirely dedicated to the

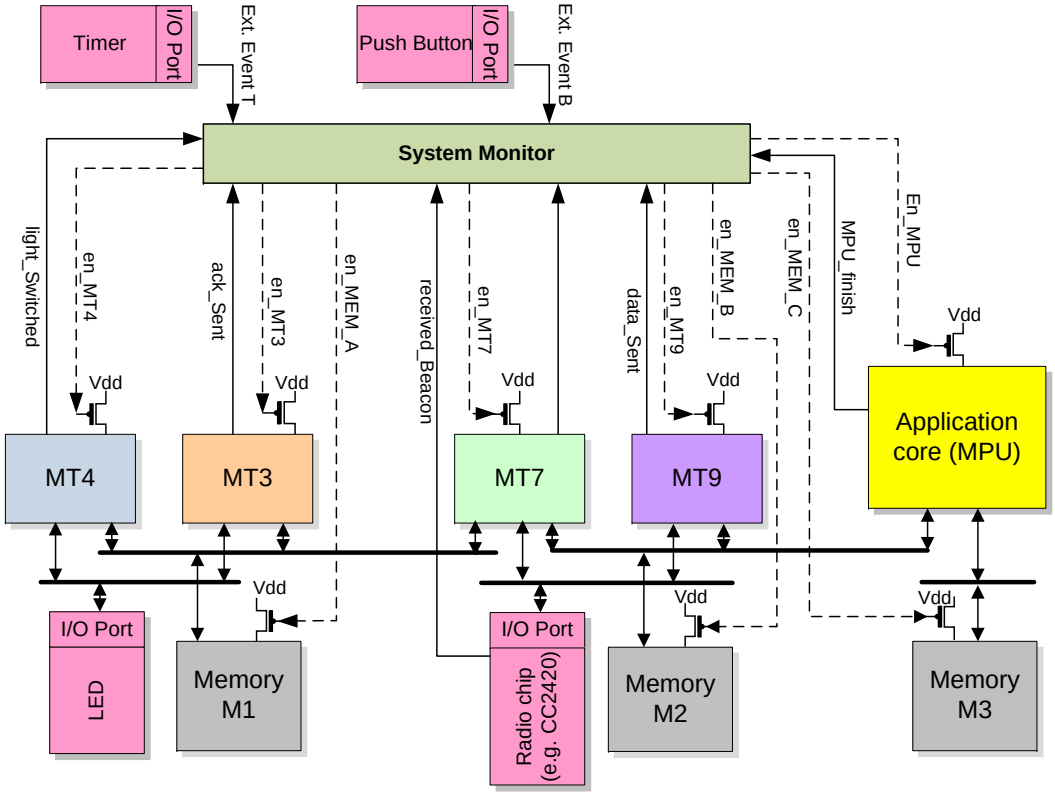


Figure 4.3.: System-level view of a micro-task based WSN node platform

communication/OS stack.

Based on this observation, we propose to combine the best of both worlds: a small silicon footprint instruction-set processor with a power-gating feature to implement the application layer software, and a distributed system of micro-tasks to handle the OS-level services of the WSN node as depicted in Figure 4.4.

This approach preserves most of the energy savings provided by specialization, while providing programmability at the application level. The system level view of such a platform is given in Figure 4.3, its main components being detailed below:

- An *application-level programmable* instruction-set processor that is used to implement the application-level code, so as to provide reprogrammability.
- A set of *power-gated application micro-tasks* accessing peripherals (RF, LED, sensor) and shared memory blocks.
- A hardware *System Monitor (SM)* that controls and schedules the execution of all the micro-tasks along with the application processor.
- Event triggering peripherals (e.g., wake-up timer) that send events to the *SM*.

A key component of the system is the *System Monitor* which takes the form of a hardwired scheduler that powers on/off micro-tasks and memory blocks and prevents concurrent access to shared resources. Because a detailed description of the SM is out of the scope of this document, we invite the reader to refer to some of our work [27] for more details.

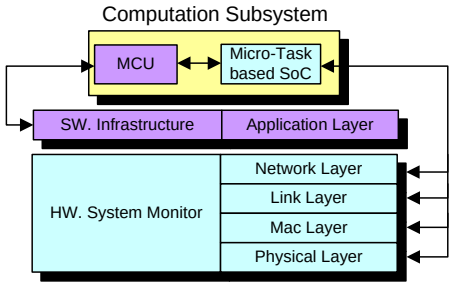


Figure 4.4.: Proposed solution to tackle the issue of loss of reprogrammability.

4.3. Microtask based System Level Design flow

Because such a platform model would be completely useless without its associated CAD tools, we have prototyped a full system level design flow, illustrated in Figure 4.5. This flow allows a WSN platform designer to derive the complete platform HDL description from (i) the description of the tasks behavior in C and (ii) a system level description of the platform expressed in a custom Domain Specific Language. In the following, we outline the main features of the flow, with a focus on the micro-task synthesis stage.

4.3.1. Synthesizing microtasks from C code

While many tools enabling the synthesis of custom hardware structure from C specifications exist, they are poorly suited for our purpose, as they focus on compute intensive kernels. On the contrary, WSN workloads mostly consist of firmware/device driver tasks, with irregular execution flow dominated by complex bit-level I/O operations.

To address the shortcoming of these tools, we designed a custom hardware synthesis flow by extending and adapting the GeCoS compiler infrastructure [105], an open source retargetable compiler framework developed in the group. GeCoS uses a simple intermediate representation (IR) in the form of a Control and Data Flow Graph (CDFG) which is used as an input to a highly flexible instruction selection framework, based on a simple [1] BURG-based tree-covering technique [69].

The originality of our approach comes from the fact that our micro-tasks are not constrained by a pre-defined processor instruction set architecture (ISA) and we can therefore leverage a relatively large number of operation patterns to obtain an efficient covering (we call this the *virtual ISA*). For example, our micro-tasks use several patterns involving complex memory operands that are common in device driver code. They also use word length specific instructions (byte, word or long operand) to efficiently determine the required bitwidth for the micro-architecture datapath. Our micro-task virtual ISA is highly customizable with minimum development efforts. As an illustration, Figure 4.6 shows some of the complex patterns that we try to match in the IR. While being complex in terms of operations, such patterns can be easily mapped to hardware, and hence are good candidates for being executed on custom functional units.

The machine-specific IR obtained through instruction selection and register allocation is then transformed into a FSM, in which each instruction is mapped to a sequence of microcode (i.e., FSM states) used to control the micro-task datapath. This transformation stage also involves a word length conversion step in which instructions operating on 16 bit or 32 bit operands may be transformed into sequential byte and half-byte level microcode. This transformation

1. There exist very sophisticated approaches for instruction selection (e.g. instruction selection of DAG patterns [102, 108]), but they are not very relevant for dynamic control dominated kernels.

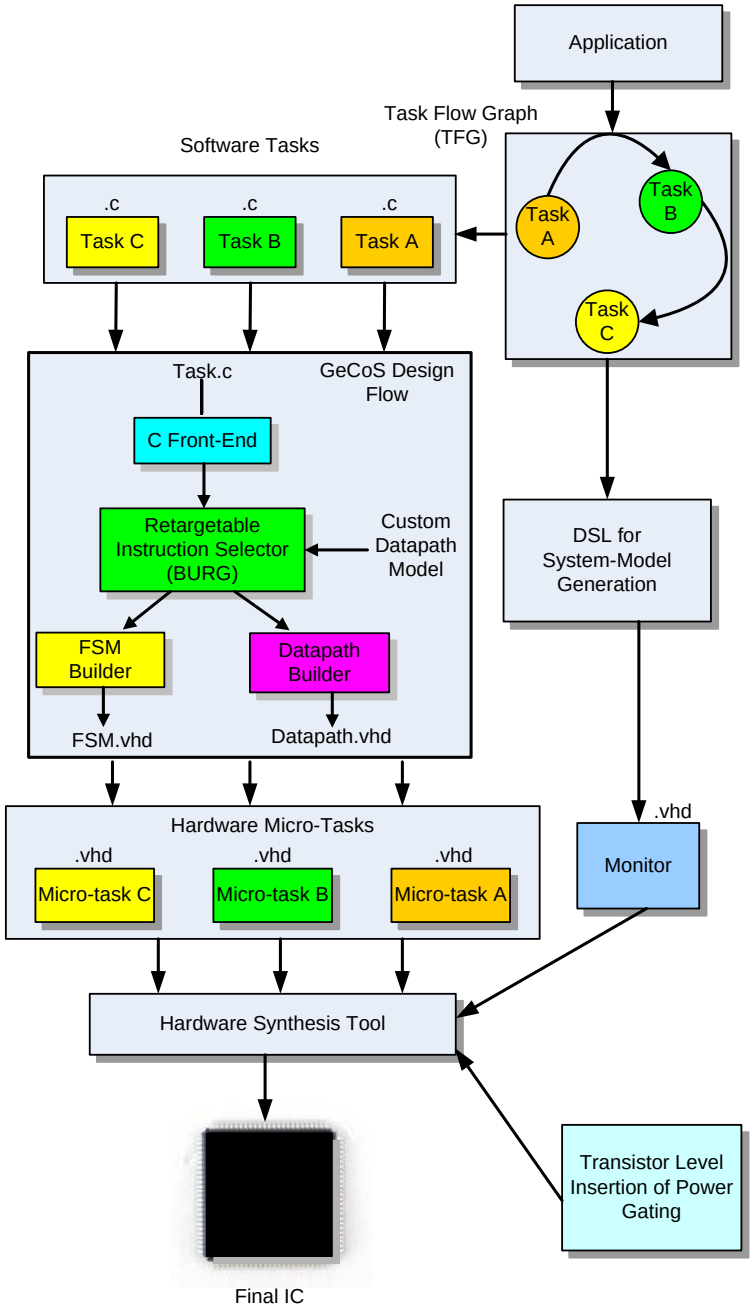


Figure 4.5.: Software design flow for the IC generation

Pattern	Action	Comments
SET(INDIR(INT), AND(INDIR(INT),INT))	andIO #ioPort, #const	**Performs an AND operation of an I/O port with a constant value**
SET(INDIR(INT), INT)	movIO #ioPort, #const	**Moves a constant value to an I/O port**
SET(INDIR(INT), OR(INDIR(INT), reg8))	orIOB #ioPort, rByte_2	**Performs an OR operation of an I/O port and an 8-bit variable**
ADD(mem, reg8)	addGB @(rByte_3), rByte_2	**Adds and stores an 8-bit variable to a memory location**
SET(GLOBAL, AND(GLOBAL, INT)	andiG @(symbol), #const	**Performs an AND operation of the memory contents pointed by a symbol to a constant value**

Figure 4.6.: Some of the rules followed in our instruction selection phase, where rByte_2 and rByte_3 are 8-bit temporaries.

helps matching the characteristics of the underlying micro-task datapath, and exploring various area/performance trade-off. This word length exploration turn out to be valuable in that it helps determining the best trade-off between performance and area for a given task. For example, the use of an 8 bit in place of a 16 bit datapath will decrease the performance and area cost but increase the power energy efficiency of the controller.

From the set of instruction patterns used in the selection phase, and after the wordlength transformation, we derive the template of the micro-task datapath. This datapath is trimmed down to provide the minimum-required functionality (types of operators and number of registers) to execute the task at hand.

The micro-task generation flow extensively uses the tools and facilities provided by the *Eclipse Modeling Framework (EMF)*, a Model-Driven Engineering (MDE) framework. More precisely, we defined a meta-model enabling the specification of complex custom micro-architectures as an assembly of a FSM and datapath components (e.g., register file, ALU operators, single-port ROM, I/O ports). We also took advantage of the template based code generation facilities provided by the Xpand [151] framework to develop a synthesizable VHDL back-end for the micro-task description.

4.3.2. System-Level Synthesis

Because most power-gating task activation/deactivation policies are difficult to express in a language such as C, we designed a Domain Specific Language (DSL) that is used to specify the system-level platform model (e.g. micro-tasks, shared memories, peripherals, etc.). As detailing the syntax of the language is out of the scope of this document, we will therefore only sketch the construction of the language by providing a simple example shown in Figure 4.7.

In this DSL, each micro-task of the system is linked to a specific C function that specifies the behavior of the task. It also specifies the event configuration that can trigger the task activation and registers the events that can be produced by a task. Similarly, the DSL is used to specify, for each task, which global variables are live (still used) or dead (e.g. their content can be lost without harm) at the end of the task execution. This information is combined with the allocation of variables and arrays to specific memory blocks in the platform.

Using the information mentioned above, it is possible to derive the complete platform description, including the System Monitor hardwired control logic. The DSL was entirely developed using MDE tools, and in particular Xtext/Xpand [151], for specifying the DSL concrete syntax and for the code generation stage.

Figure 4.5 shows the complete design-flow for micro-task based node generation: it takes as inputs the application description modeled as a TFG using the DSL, and each tasks written in C, outputs VHDL code for the node.

4.4. Experimental results

Providing a strong quantitative analysis of the approach through benchmarking is crucial for our work. However, even if there has been several attempts to profile the workload of a generic WSN node, only few of these research results are actually usable in our context.

We selected our benchmarks from the two benchmarks for WSNs workloads that we are aware of: SenseBench [116] and WiSeNBench [115]. We also added additional tasks taken from PowWow, an open-source WSN platform [84] developed in the group. The synthesis was performed for both 130 nm and 65 nm CMOS technologies using *Design Compiler* from Synopsys.

```
system send_receive_data {
	include "send_receive.gecos"
	events { extPB, extET, beacon_Sent, data_Received,
			ack_Sent, timeOut0, timeOut1, timeOut2, receiver_OFF, transmitter_OFF,
			counter_Start, beacon_Received, data_Sent, ack_OK, ack_NOK, radio_OFF}

	memory memB [gated] {
		contains globals {neigh_IdX,neigh_IdY, receiveFrame, sentFrame, pushButtonStatus}
	};

	memory memC [permanent] {
		contains globals {my_IdX, my_IdY}
	};

	ioModule led {
		contains ports { port LED 8}
	};

	ioModule pushButton {
		contains ports { port PUSHBUTTON 7}
	};

	ioModule cc2420 {
		contains ports { port P2IN 0, port P5OUT 1, port U1TCTL 2, port U1RXBUF 3,
					port U1TXBUF 4, port URXIFG1 5, port IFG 6}
	};

	microTask receiveData {
		activates With { beacon_Sent }
		produces { data_Received }
		reads ioModule { cc2420 }
		writes memory { memB }
	};

	microTask sendBeacon {
		activates With { extET }
		produces { beacon_Sent }
		writes ioModule { cc2420 }
		reads memory { memC }
	};
}
```

Figure 4.7.: An example of the system level DSL.

Task Name	8-bit Micro-task											
	No. States	time (μs)	Power (μW)		Energy (pJ)		P. Gain (x) P1/P2		E. Gain (x) E1/E2		Area (μm^2)	
			130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm
crc8	71	4.4	30.09	8.0	132.4	35.3	292/32	1095/32	339/37	1272/37.3	5831.7	1762
crc16	103	6.4	46.92	12.4	300.3	79.2	187/20.4	710/21	140.5/15.3	532.8/15.5	8732.5	2678
tea-decipher	586	36.6	84.5	22.6	3090	827	104/11.4	389/11.3	78/8.55	292.6/8.6	19950	6138
tea-encipher	580	36.2	87.3	23.3	3160	845	101/11	377/11	75/8.2	281/8.3	20248	6230
fir	165	10.3	75.3	20.4	775.6	209.7	116/12.8	432/12.5	123.8/13.4	458/13.3	13323.7	4124
calcNeigh	269	16.8	74.3	20.1	1248.2	337.8	118/12.9	437/12.7	142.4/15.5	526/15.4	14239.4	4454
snd2SPI	672	42	33.3	8.84	1400.3	371.3	264/28.8	995/29	198.5/21.7	748/21.7	10578	3434
rcvFromSPI	332	20.7	27.3	7.4	565	153.2	322/35	1189/34.6	247.6/26.7	913/26.8	5075.3	1561

Table 4.2.: Power and energy gain of 8-bit micro-tasks over MSP430 (@ 16 MHz). Here, P1 and E1 are the power and energy gains w.r.t. tiMSP whereas P2 and E2 are the power and energy gains w.r.t. openMSP.

Task Name	16-bit Micro-task											
	No. States	time (μs)	Power (μW)		Energy (pJ)		P. Gain (x) P1/P2		E. Gain (x) E1/E2		Area (μm^2)	
			130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm	130 nm	65 nm
crc8	71	4.4	55.3	14.71	242.6	64.72	159.6/17.4	598.2/17.4	185.1/20.2	693.7/20.3	10348	3097
crc16	73	4.56	55.0	14.69	251.0	66.98	159.8/17.4	599/17.4	168.1/18.3	630/18.4	10280	3102
tea-decipher	308	19.2	152.8	40.85	2940	784.3	57.6/6.2	215.4/6.3	82/9	308.5/9.04	27236	8380
tea-encipher	306	19.1	152.3	40.61	2910	776.0	57.8/6.3	216.7/6.3	81/8.93	306.2/9	27069	6211
fir	168	10.5	144.2	39.03	1514	409.8	61.02/6.7	225.5/6.56	63.4/6.9	234.3/6.8	23547	7164
calcNeigh	269	16.8	142.4	38.58	2392	648.1	61.8/6.7	228/6.4	74.3/8.1	274/8	24745	7613
snd2SPI	672	42	58.1	15.53	2440	652.2	151.5/16.5	566.6/16.5	114/12.4	426/12.52	14863	4771
rcvFromSPI	332	20.7	50.0	13.67	1036	283.0	175.8/19.2	643.7/18.72	135/14.7	494/14.42	9485	2858

Table 4.3.: Power and energy gain of 16-bit micro-tasks over MSP430 (@ 16 MHz). Here again, P1 and E1 are the power and energy gains w.r.t. tiMSP whereas P2 and E2 are the power and energy gains w.r.t. openMSP.

Power and area estimations are given in Tables 4.2 and 4.3. As expected, the power dissipated by FSMs increases with their number of states, following a sub-linear relation. As a result, an 8 bit micro-task consumes nearly half the power and silicon area of a 16 bit micro-task, suggesting that the FSM of a micro task consumes much less power than its associated datapath. For tasks mostly involving word lengths greater than 8 bits, the total energy consumption of an 8 bit and 16 bit micro-task is nearly the same. On the other hand, for applications where 8 bit word length operations dominate, an 8 bit micro-task consumes half the energy of a 16 bit micro-task.

We also synthesized the *openMSP*-core for 130 nm process to get an estimate of its silicon footprint, which is 75000 μm^2. To evaluate the area overhead of our micro-task based decomposition, we took as a baseline the cumulated area cost of all the microtasks of our case study, that is roughly 48000 μm^2 (2/3 of the openMSP) for the same technology. This shows that our approach is also competitive in terms of hardware resource usage. Of course, our main concern is energy savings, and we also evaluated the savings that could be obtained against off-the-shelf MCUs by considering two different versions of the MSP430:

- *tiMSP*, a TI MSP430F21x2 using the datasheet information (8.8 mW @16 MHz in active mode) which includes memory and peripherals,
- *openMSP*, an open-source MSP430 processor core without accounting for program and data memory. Statistical power for 130 nm technology was estimated to be 0.96 mW @ 16 MHz.

The exact energy-efficiency in terms of *Joules/instruction* of micro-tasks cannot be measured,

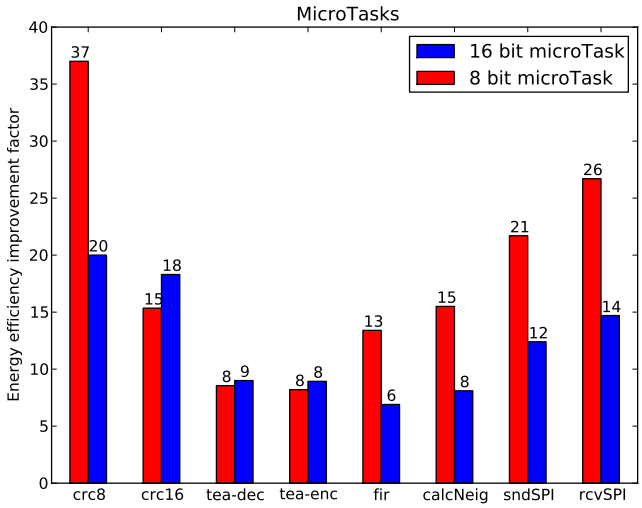


Figure 4.8.: Energy improvements factors offered by 8 and 16 bits micro-tasks in 130nm w.r.t. to the openMSP baseline.

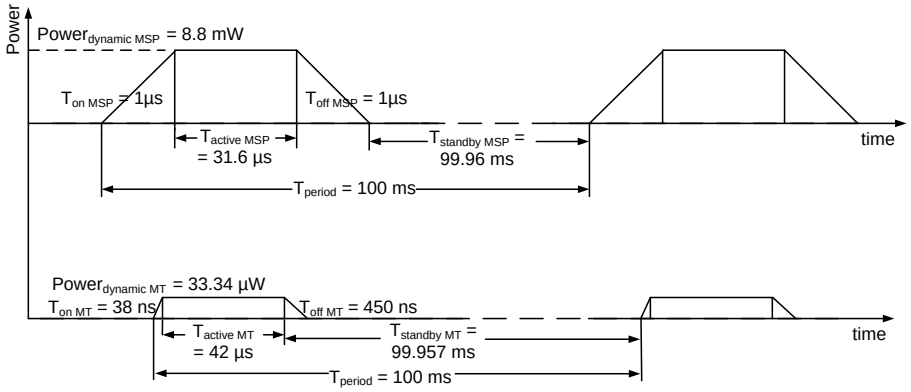
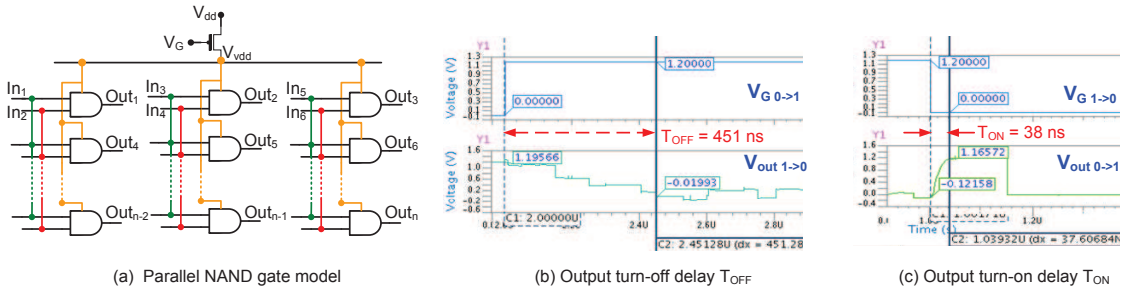


Figure 4.9.: Time distribution of *snd2SPI* task duty cycle.

as micro-tasks are not instruction-set processors. Hence, we used a notion of *Joules/task* as used by Hempstead et al. [80] in their work. Since these micro-tasks are comparable to the MSP430 in terms of execution time, we used the instruction count for MSP430 implementation and the actual energy consumption of the hardware micro-task (for each application and control task) to estimate equivalent energy efficiency.

The results in terms of energy efficiency improvement (for the openMSP) are summarized in Figure 4.8. They show significant energy efficiency improvements, ranging from a factor of 8 (in the most pessimistic case) to more than 35 (for the optimistic case). The results are roughly ten time better for the TI-MSP430 as the openMSP core is ten times more energy efficient than the latter (as a matter of fact, we expect the actual energy efficiency of the full MSP430 with its memory to lie somewhere in between these two results). For the sake of completeness, and to show that its impact on the energy budget is limited, we also synthesized the *System Monitor* for our lamp-switching WSN node. Results confirmed our expectations, as the later dissipates a mere 12 μW at 16 MHz.

Even though the above result are very encouraging, they only show the dynamic power savings obtained through specialization. In the context of WSN with low duty-cycles (as illustrated in Figure 4.9), static power plays an important role in the global power budget. In particular,



(a) Parallel NAND gate model (b) Output turn-off delay T_{OFF} (c) Output turn-on delay T_{ON}

Figure 4.10.: Time distribution of *snd2SPI* task duty cycle.

the ability to reduce the sleep and wake up delays from/to power gated mode can also contribute to energy reduction, as components continue dissipating dynamic and static power during this transient state.

To obtain accurate timing information, we have performed transistor level simulation using a simple load model for micro-tasks (see figure Figure 4.10.(a)). The results of these simulations are shown in Figure 4.10.(b) and Figure 4.10.(c). The estimated turn-on and turn-off delays (between cut-off and active mode) are 38 ns and 451 ns for a 3000 gate equivalent component (a representative area cost for a micro-task).

Considering this static power, a chosen time period of 100 ms and by using a simple analytical model (see the full paper [27] for more information) we calculated the overall energy saving for *snd2SPI* micro-task over a complete period of task-activation. For this representative example the combined use of specialization and power gating lead to an energy reduction of a factor of 138 over the MSP430, considered as a reference for its extremely low static power dissipation.

4.5. Discussion

The design flow presented in this chapter is a complete end-to-end flow, on which we started working in late 2007 with the PhD thesis of Adeel Pasha. We first started studying the problem at the transistor level, to first evaluate the relevance of power-gating for fine to medium grain circuits. As the results that we obtained were promising, we decided to prototype a complete software flow starting from so-called system level specification [26] down to synthesizable hardware [25].

This prototype helped us obtaining more solid quantitative evidence of the relevance of the approach, and lead to the publication of four articles in conferences [26, 25, 24, 28], including a paper presented at the IEEE/ACM Design Automation Conference in 2010. A journal article has also been written and was accepted in July 2011 in the ACM Transactions on Design Automation for Embedded Systems. The work done in this PhD is being continued by Vivek Tovinakere, who is currently addressing the problem of providing accurate high-level analytical timing and energy models for power gated blocks, to avoid complex and long transistor level simulations [5, 6].

Designing a complete design flow in less than two years was definitely a challenge, especially given the fact that the PhD student who did most of the implementation work [2] had little experience in programming an no notion of compiler design. Even if this work did not lead to outstanding contributions in terms of new synthesis techniques and/or algorithms (it was not the topic of the work), it still looks to me like an achievement.

2. an Electrical Engineering major.

I believe part of this success was made possible by systematic use of Model Driven Software Design, which helped us a lot in formalizing all the models (system-level, RT-level, etc.) used in the flow and provided us with the adequate facilities for building their associated toolset (parser, code generators, etc). This success also demonstrates that even non computer science experts can (and should) make use of these technologies, which work both as productivity and creativity boosters.

Summary	
	Supervision: Adeel Pasha, graduated in 2010, 70 %, now assistant professor at University of Engineering and Technology, Lahore (Pakistan)
	Journal: ACM Transactions on Design Automation for Embedded Systems
	(accepted for publication in July 2011) [27]
	Conferences: IEEE/ACM Design Automation Conference 2010 [25]
	Euromicro Conference on Digital System Design 2010 [26]
	IEEE International Symposium on Circuits and Systems 2009 [24]

Chapter 5

Synthesis of Hardware accelerator for regular computations

The contributions presented in this chapter are spread over a relatively large time frame (from 2003 to 2011). They still tackle the same problem: how to automatically synthesize parallel custom hardware from representations of programs based on the polyhedral model.

The chapter is organized as follows, we will first start by a short summary of the principles behind the polyhedral model, along with an overview of existing techniques and tools used for the synthesis of custom hardware from such representations. This survey is followed by a detailed description of our three contributions, each of them being related to a distinct approach. To the contrary of previous chapters, we discuss the results of our approaches in each section.

5.1. Representing loop nests as polyhedrons

Regular and repetitive computation patterns (such as those found in nested loops) are known to be interesting candidates for hardware acceleration, as (i) they often represent the compute intensive kernels in a program and (ii) their regular structure makes them amenable to advanced analyses and transformations, such as those offered by the polyhedral model.

These analyses and transformations techniques leverage on theoretical foundations that date back to the early 80s. However, this model only recently found its way within optimizing and parallelizing compilers. This increasing adoption is mainly due to a recent renew of interest for automatic parallelization techniques, mainly motivated by the outbreak of muti-core architectures. This revival led to many breakthroughs (both practical and theoretical) over the last few years [39, 44, 124, 125, 35].

The core idea behind the polyhedral model is to provide a compact *iteration wise* representation of static control loop nests using integer polyhedrons. The approach supports imperfectly nested loops with affine array accesses and affine guards (Static Control Parts [40]), although recent work has addressed its extension to wider classes of programs [43, 51]. In the following we will call SCoPs such static control parts of programs.

The polyhedral model enables powerful analyses and transformations ranging from data locality optimizations to automatic parallelization. The model provides a unified framework in which it is possible to express (and check the legality of) complex combinations of loop transformations, leading to much more efficient generated code than those obtained by combining independent transformations [71].

In the polyhedral model, all program transformations are expressed as *affine* (or *quasi affine*) transformations of the statement domains. The transformed program is obtained through a code generation phase that reconstructs the new loop nests from the transformed domains. Figure 5.1 illustrates such a transformation, where the chosen schedule expresses a combination of a *loop interchange* transformation followed by a *loop strip-mining*. The reader may notice that in this new program, the innermost loop can be easily vectorized as it does not carry dependencies.

Although most of the existing tools and techniques are geared toward general purpose and/or parallel machines, they can also be used in the context of hardware synthesis, as explained in the following subsection.

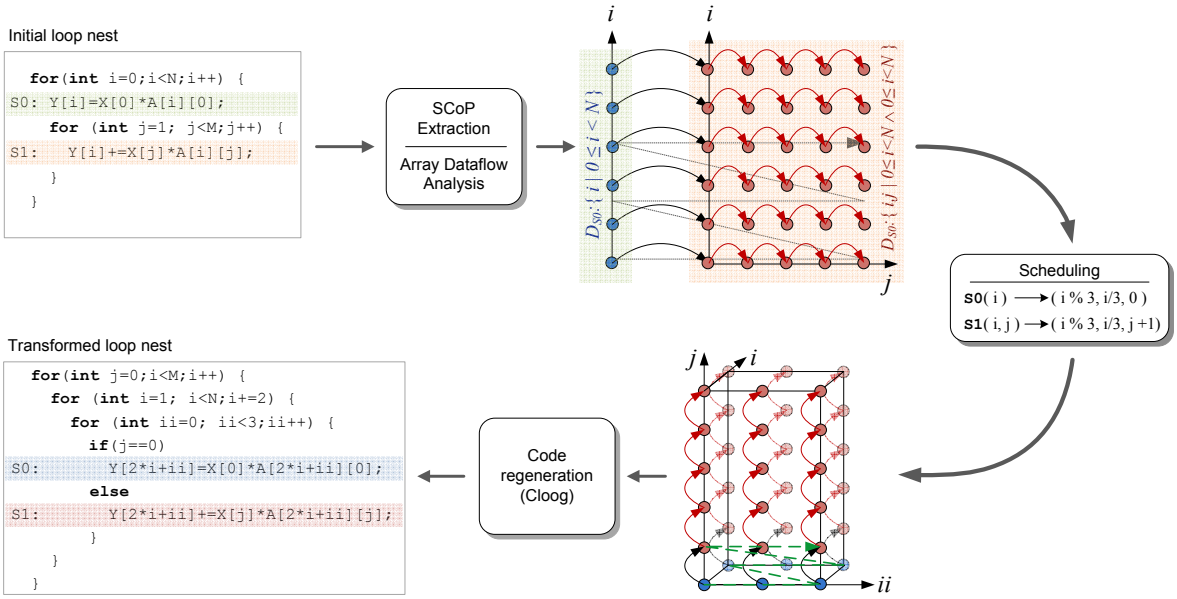


Figure 5.1.: Illustration of a polyhedral based loop transformation flow. Each statement is associated to a polyhedral domain. In the figure, dependencies between statement instances are represented by arrows.

5.2. From processor arrays to process networks

There exist several different approaches (and tools) for synthesizing nested loops onto custom hardware accelerators. Figure 5.2 provides an overview of the different approaches, detailed below.

Synthesizing Regular Processor Arrays

The idea of exploiting the regularity of loop structures to derive parallel architectures is not new, and this research problem has received a lot of attention in the 80s thanks to the seminal work of Kung and Leiserson in 1978 [94]. This led to the concept of *systolic arrays*, later generalized as *regular processor arrays*, that can be seen as a regular multidimensional mesh of very simple processor cells, supporting a nearest neighbors communication topology.

The work on processor arrays also led to important contributions in automatic parallelization [126, 161], and design automation [163], which led to semi automatic design tools such as MMAlpha [74] and PARO [60, 77]. Many important applications in the field of signal processing and linear algebra have shown to be well suited systolic arrays, and there exist some application domains (in particular in wireless communications) where (small scale) processor arrays are implemented as building blocks of more complex systems. However, it turns out that processor array architectures (and their supporting design methodologies) suffer from important limitations:

- Experience has shown that automatically determining a mapping and a scheduling that would be competitive with a manual design is a difficult (and still open) problem.
- The constraints on the architecture are often too restrictive and limit the set of applications and/or kernels amenable to efficient mappings.
- The architectures derived using those methodologies are poorly suited to a mapping on heterogeneous multi-processor platforms.

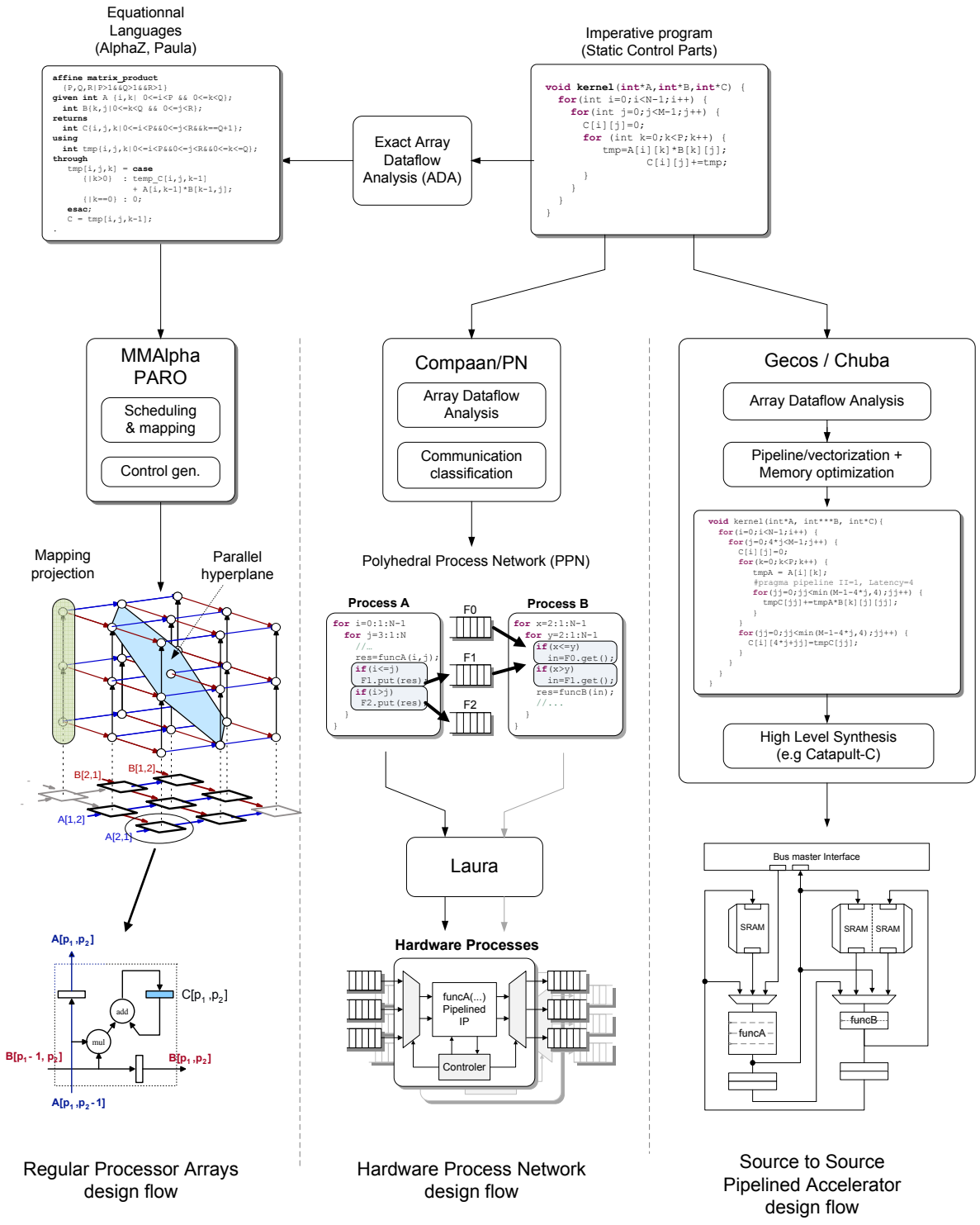


Figure 5.2.: An overview of existing hardware synthesis flows based on the polyhedral model

Among these limitations, the later two are probably the most severe, and hence motivated the search for alternative approaches, better suited to current hardware platforms.

Deriving Hardware Process Networks

Rather than trying to map the computation to a processor grid, the Compaan framework [153, 90, 152] instead proposes to produce a Process Network representation of the program, known as Polyhedral Process Networks (PPN) [154]. In this approach, operations of the program are mapped to different concurrent processes, communicating through FIFOs enforcing Kahn Process Networks semantics [86] (blocking read, non blocking write). This model is much more suited to an implementation on heterogeneous multi-core platform, as it focuses on more coarse grain (task-level) parallelism and does not require shared memory.

An interesting property of PPN is that deriving such a model does not involve determining a (legal) global schedule for the computations, something known to be difficult. Instead, the PPN relies on data-flow synchronization (through FIFO buffers and local reordering memory) to ensure that a process firing cannot happen before all its operands are available.

However, this flexibility is also one of the main weakness of the approach. Because every statement of the program is mapped to its own process, the resulting process network is highly dependant on the initial input program specification (and its implicit underlying schedule). This may lead to network sizes that are not suited to the target platform and/or requirements (e.g., too many nodes will induce too much scheduling overhead, while to few nodes may underutilize the platform resources). A lot of work has therefore been carried on process splitting/merging transformations [109, 144], so that the designers have tighter control on the characteristics of the generated network.

Optimizing Pipelined Accelerators

A more recent trend aims at taking advantage of the huge progress in High-Level Synthesis in the last ten years, and also of the large choice of robust and mature tools that nowadays exist [73, 30, 32].

All these "C to hardware tools" significantly slash down design time. However their ability to generate efficient accelerators is still limited, and relies on the designer to expose parallelism and to use appropriate data layout in its source program. Therefore, there is a growing interest for source-to-source compilers, which could be used as front-end automatic parallelization tools, to derive heavily pipelined (and possibly vectorized) hardware accelerators. This is the case for Gecos-S2S4HLS [130] and Chuba [123] frameworks, which aim at providing a polyhedral based loop transformation toolbox for improving the efficiency of HLS tools for nested loops.

Summary of the contributions

All contributions presented in this chapter are related to one of the three approaches depicted in Figure 5.2.

The first work presented in this chapter (Section 5.3) is in the direct continuation of my PhD, which focused on the synthesis of efficient processor arrays, through the use of *partitioning transformations* [12, 15, 14, 11, 13] and on automatic interface synthesis for linear processor arrays [16, 8]. The goal of this work was to study how to extend our previous results to handle the derivation of efficient I/O interfaces for arbitrary dimensions partitioned processor arrays.

The second contribution (Section 5.4) was a result of my stay as a post-doc in 2003 at Leiden University in the group of Ed Deprettere. During this period, I continued working on the problem of automatic hardware synthesis, this time in the context of the Compaan/Laura

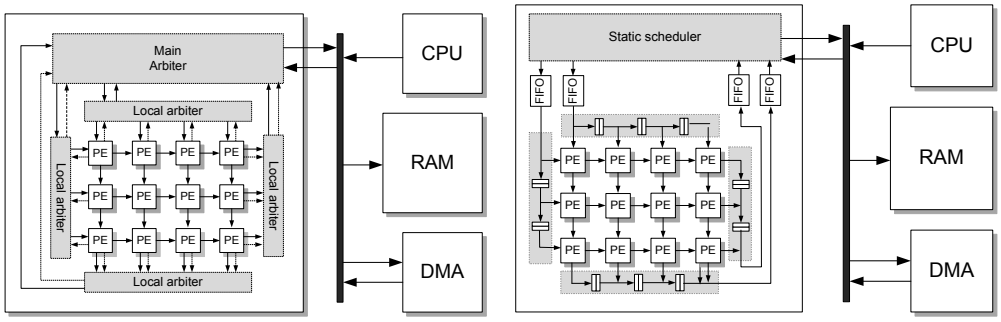
framework [165, 153]. More precisely, the goal was to synthesize efficient hardware process networks [18] following the Polyhedral Process Network operational semantics [154].

Since late 2009, I have been working, in the context of the Nano 2012 research program between Inria and STMicroelectronics, on nested loops source-to-source transformations for High-Level Synthesis tools for GeCoS compiler infrastructure. The topics addressed in this work revolve around nested loop pipelining and efficient code generation and are developed as part of the PhD thesis topic of Antoine Morvan.

5.3. Efficient I/O management in processor arrays

The high-level synthesis research community has mostly focused on deriving efficient dedicated hardware accelerators from high-level specifications. The problem of automatically generating interfaces between these accelerators and the rest of the hardware system has received only little attention.

However, as most designers can tell, such interface is often a very tedious and error-prone part of a design. Moreover, a poorly designed interface can drastically reduce the actual accelerator performance. This problem is even strengthened for stream processing applications; huge parallelism present in the accelerator can be ruined by an inefficient handling of data-stream communications.



(a) Dynamic I/O management (b) Static I/O management

Figure 5.3.: Two different approach for interfacing 2D processor arrays

In this work, we tackled the problem for multi-dimensional *partitioned processor arrays*. Pure systolic architectures are impractical because of the huge throughput and resource requirements. As a consequence, several strategies have been proposed to derive processor arrays under resource or I/O constraints [137, 112, 85, 53, 52, 55], leading to a deep understanding of the *partitioning* transformation [149, 67, 55, 15, 12].

However, these approaches did not address the interface issue, which is how to efficiently integrate the resulting architectures within a complete system. This problem was studied in the case of linear processor arrays [120, 121, 8, 68], where data is entering the array from a single processor element. However the problem of multi-dimensional processor arrays did not receive much of attention, but from the PARO project [42] and the Pico tool [135].

Interfacing multi-dimensional arrays is more difficult than 1D arrays because many data can enter the array simultaneously. At some point, this data have to be sequentialized in a FIFO-like channel that can be connected to a memory through a bus or a network on chip with reduced scalability.

By carefully choosing the partitioning parameters, the designer can adapt *on average* the bandwidth required by the processor array to the bandwidth available on the communication

medium [12]. However, there is no guarantee that two processors of the array will not access the bus simultaneously. A natural solution is to implement a dynamic resolution of the conflicting accesses with an arbitration mechanism, as suggested in [135] and as illustrated in Figure 5.3(a).

In this work, we have proposed an alternative solution by showing that a static schedule without conflicts can be found for I/Os of partitioned array processors. More precisely we propose a technique to derive a conflict free I/O pipeline along the processor array boundaries. Thanks to this static I/O schedule, the hardware area can be made smaller by using small shift registers rather than dynamic arbiters, as shown in Figure 5.3(b). Moreover, the fact that the I/O schedule is known in advance enables more advanced burst mode communications and prefetching techniques. Our methodology was experimentally validated by a VHDL implementation of a partitioned matrix-product array, and its results compared to a bus arbiter based approach.

5.3.1. Conflict free I/O schedules in partitioned processor arrays

To illustrate the techniques used in this work, it is necessary to briefly describe the main ideas behind *processor partitioning* techniques. The goal of *partitioning* is to reduce the level of parallelism in processor arrays by *partitioning* the original processor array into parallelepipedic *tiles*, using a family of hyperplanes. Two partitioning strategies have been proposed, which can be combined through a hierarchical partitioning approach.

- Locally Parallel Globally Sequential (LPGS) [111] partitioning, consists in executing all computations within a partition in parallel and atomically. This execution is performed on a smaller processor array, whose size corresponds to the partition size.
- Locally Sequential Globally Parallel (LSGP) [48, 52, 54] partitioning consists in executing the partitions concurrently, by using a single processor per partition, which scans all the iterations within that partition.

In this work, we are interested in the LSGP scheme. LSGP is a very efficient transformation to reduce the I/O bandwidth of the processor array, as the I/O throughput decreases as the size of the partition grows. The LSGP transformation is illustrated in Figure 5.4 and Figure 5.5, where the partition is a simple 2×3 rectangle. In this example the three processors along the vertical axis perform an I/O only once every three cycles instead of every cycle in the original array, hence matching (in average) the FIFO throughput.

However, as depicted in Figure 5.4, if we assume that data is broadcasted from the FIFO to each processor, the resulting schedule may lead to some conflicts, as several processors need to access the FIFO output at the same time instant. On the other hand, if we assume another communication mechanism, for example a pipeline in reverse order of the processors vertical axis (with one register per processor) as in Figure 5.5, the I/O schedule is now conflict free.

We proposed a method to determine automatically the set of legal (i.e, conflict free) data pipelines, by expressing constraints on the pipeline direction (forward or reverse) and on the number of registers between processors. These informations are derived from both the processor array initial schedule and partitioning parameters.

5.3.2. Experimental results

To illustrate the benefits of our approach, we compared our interface implementation to an alternative one based on a run-time management of the I/O conflicts. In this case, whenever two or more processors have to perform an I/O for the same stream at the same time, a hardware arbiter is used to select the access to be scheduled first. We prototyped (in VHDL) the two approaches for a matrix-matrix product. The choice of this basic example does not affect the

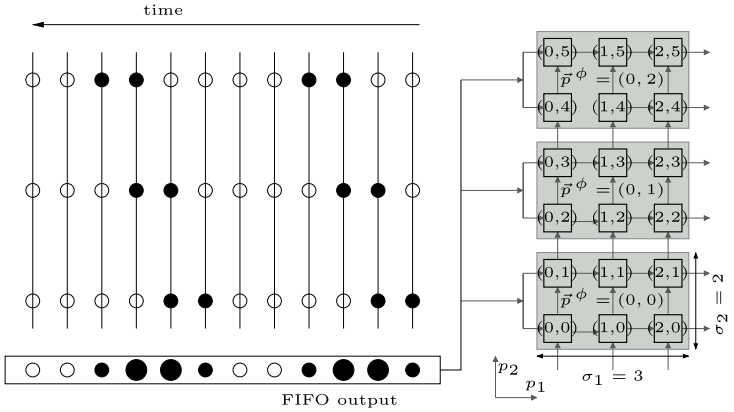


Figure 5.4.: Conflicting I/O schedule: at time $t = 1, 2, 7, 8$, several processors access the FIFO (conflicts shown as large black circles).

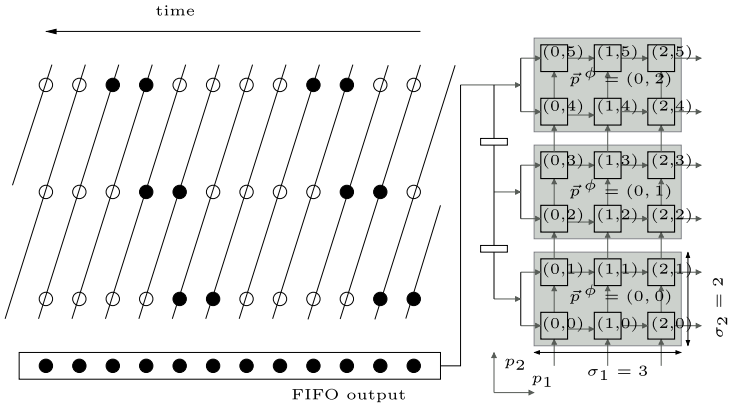


Figure 5.5.: The I/O schedule is dense (FIFO accessed every cycle) and conflict free (at most one access to the FIFO at a given cycle).

Matrix size	Physical array shape	Area				
		static			run-time	
		LUT/DFF/SRL16			LUT/DFF	
$32 \times 32 \times 32$	2×2	116	96	128	285	266
$32 \times 32 \times 32$	4×4	116	96	256	539	508
$64 \times 64 \times 64$	4×4	132	102	256	603	532
$64 \times 64 \times 64$	8×4	132	102	320	784	672
$64 \times 64 \times 64$	4×8	132	102	448	1048	888
$128 \times 128 \times 128$	8×8	156	108	512	1454	1160
$128 \times 128 \times 128$	8×4	156	108	320	970	768
$128 \times 128 \times 128$	16×8	156	108	640	1811	1472

Table 5.1.: Experimental results for interface area: number of logic cells (LUT), of registers (DFF), and of Xilinx shift registers primitives (SRL16) (not used in the run-time approach).

validity of the results, as any two dimensional processor array will use a similar interface [1].

For both approaches, we synthesized a set of architecture instances that only differ by some parameters. A summary of the results is given in Table 5.1; to make a fair comparison, one should compare the sum of LUT and SRL16 (for the static interface) to the number of LUT in the run-time approach, and the DFF for both approaches. Experiments show that our approach based on a static I/O schedule leads to significant area savings: while the number of flip-flops (DFF) required to implement the interface is roughly the same in both approaches, the number of LUT resources is much lower in our approach especially for larger processor arrays.

It is also worth noticing that the selected values for the r_j parameters do not affect the area cost of the interface. This is due to the use of the shift-register primitives provided by Xilinx architectures, which allow shift-registers; up to 16-bit deep; to be packed into a single logic cell.

5.3.3. Discussion

I started thinking about this problem during my PhD thesis, but had to wait until 2004 to actually to start working on it. After several discussions with Tanguy Risset, and after we realized that the problem could benefit from Alain Darte formalization of the so-called *Juggling* problem, we started working together on the topic. Interestingly, a work addressing a problem similar to ours was also published in 2005 (one month before ours) by Hannig et al. [78].

In retrospect, the main weakness of this work is that it remained theoretical, in the sense that it was never implemented in the MMAlpha framework since the MMAlpha did not support Z-polyhedral domain (a prerequisite for managing partitioned processor arrays).

The questions and issues raised by this work (in particular the idea of reordering buffer memory) however led the co-authors of this work (Alain Darte and Tanguy Risset) to start a PhD on the topic (PhD Alexandru Plesco) at LIP in Lyon. However, they tried to tackle the problem from a higher abstraction level by a combined use of High-Level Synthesis tools and source-to-source transformations [123].

Summary	
	Conferences: IEEE ASAP 2005 [7]
	Collaborations: LIP - ENS Lyon

1. Things are slightly more complicated for three dimensional arrays and above, see the full paper for more details [7].

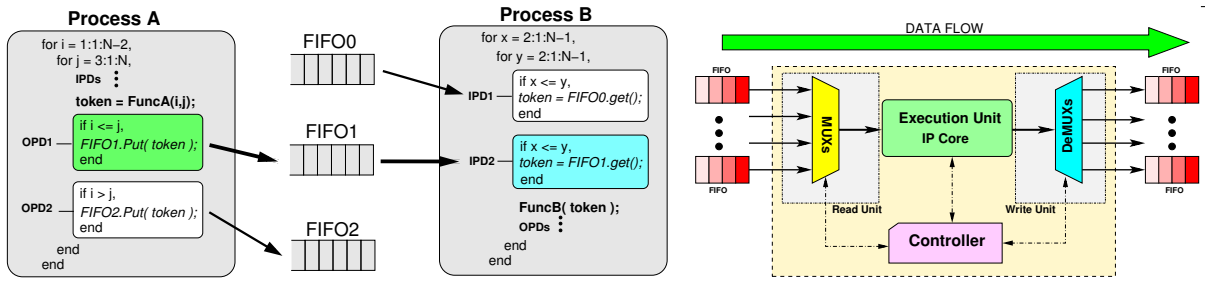


Figure 5.6.: (a) Two processes taken out from a Compaan generated PN, (b) Hardware realization of a process

5.4. Control generation for hardware Process Networks

As mentioned in the beginning of the chapter, the goal of the Compaan framework [89, 145, 132] is to automatically transform applications in the field of signal and image processing to Process Networks. Compaan operates on a small subset of Matlab and produces a Polyhedral Process Network [154] model of the program.

In Figure 5.6(a), we show a fragment of a process network obtained from Compaan, where processes A and B communicate data over FIFO channel FIFO1. In both processes, nesting of for-loops indicate the schedule in which the functions **FuncA** and **FuncB** are being executed, affine guards determine which FIFO is used as source or sink.

The Laura tool serves as a back-end and transforms the process network into a synthesizable VHDL description. During this conversion, each process is mapped to a *hardware process*, as shown in 5.6(b), whose schedule is handled by a local hardware controller. Because of the inherent complexity of the schedules handled by Compaan, deriving a low footprint and fast hardware realization for this control is mandatory for obtaining an efficient and practical implementation of this processor.

The initial implementation of the control in Laura was based on ROMs, which would contain a compressed control sequence of an *unrolled* execution of each process, using a simple Run Length Encoding Technique. Some results obtained with this technique for a few representative applications are given in Table 5.2. In spite of its relative efficiency for small iteration domains, the technique does not scale for large and/or complex domains (e.g., those that can be found in image processing applications).

	N	T	direct bytes	RLE bytes	% of mem FPGA
QR	16	64	7680	4160	0.9
(node 4/5)	64	256	516096	78080	17.12
	W	H	simple	RLE	% of mem
Stereo vision	640	400	1941576	1698240	372.42
(node 4/5)	1024	640	5072328	4437264	973.08
Optical Flow	640	480	3808860	29850	6.54
(node 3/7)	1024	764	9780540	47850	10.4

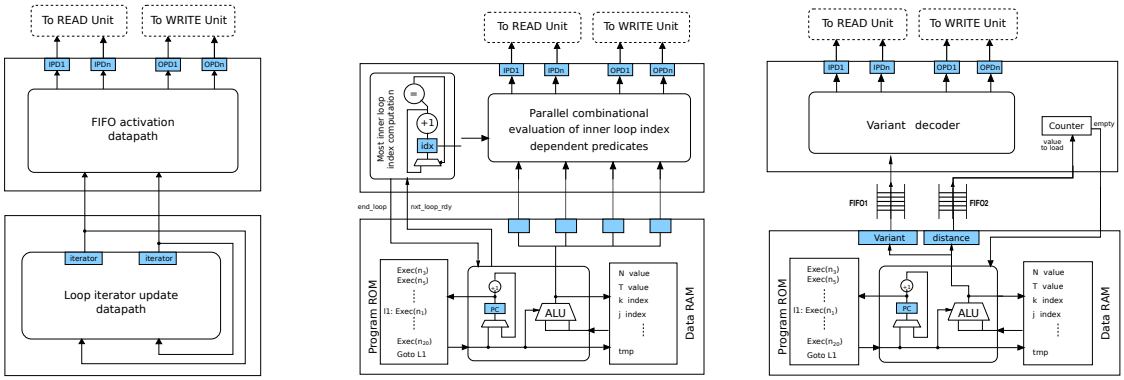
Table 5.2.: Control ROM size for three different applications

This work has hence consisted in investigating different strategies for deriving hardware realizations for the controller and in exploring the trade-offs between speed and resources usage.

5.4.1. Simple parameterized Controller

The first approach that we studied consists in building an optimized dataflow graph out of the guard and loop bounds expressions available in the process control flow. This graph is then directly mapped onto a simple datapath structure, following the template given in Figure 5.7(a), where each operation is mapped to its own operator.

Because of the class of nested-loop programs that Compaan accepts, all predicates and bounds consists of *quasi affine expressions* of the loop indices and parameters. These expression offer many simple optimizations opportunities. In particular, we combined *redundancy elimination* to remove redundant operations and *bit accurate wordlength analysis* using Integer Linear Programming to determine the exact number of bits needed to encode the actual range of loop indices.



(a) Simple controller (b) Partitioned controller (c) Variant based controller

Figure 5.7.: The three types of controller under study in this work: the parameterized simple controller, the partitioned controller and the variant based controller

5.4.2. Partitioned parameterized Controller

The parameterized controller, depicted in previous subsection, is sensitive to the amount of linear expressions that need to be evaluated in each iteration. To reduce the amount of computations, we use the fact that only expression depending on the innermost loop index have to be evaluated at every cycle. Following this observation, we split the control in two different blocks: a parallel datapath that evaluates all expressions depending on the *innermost loop-iterator* and a sequential micro-coded controller for the remaining ones, as depicted in Figure 5.7(b).

The architecture of this controller is derived in two steps. The *DAG partitioning* step first isolates all subgraph patterns that do not depend on the innermost loop index. This step is followed by a scheduling step, that computes the schedule to derive the micro-code of the sequential controller. The parallel datapath is derived using the same principle as in the Simple Parameterized Controller approach, except that all arguments depending on values calculated by the sequential controller are mapped to communication ports.

To observe the benefits of our approaches, we used the same benchmarks as in Table 5.2. Results are given in Table 5.3 and show that the partitioned approach is only useful for large and complex domains, and induces a (small) performance penalty (in terms of clock speed).

	Size		Non partitioned		Partitioned	
	N	T	MHz	Area	MHz	Area
QR factorization	8	16	140	29	100	112
	16	64	133	68	85	133
	64	256	121	89	74	163
	W	H	MHz	Area	MHz	Area
Stereo-vision	320	200	97	133	65	120
	640	400	100	148	74	123
	1024	640	100	153	71	126
	W	H	MHz	Area	MHz	Area
Optical-flow	320	200	129	97	76	98
	640	400	118	110	72	103
	1024	640	126	113	75	106

Table 5.3.: Experimental results for three representative examples.

5.4.3. Toward a Variant based controller

Looking at the code in Figure 5.6(a), one can notice that, in most cases, a given configuration of active FIFO will stay the same for a certain number of consecutive iterations. In the following, we call such a configuration a *variant* as proposed by Kienhuis [88]. Taking advantage of this regularity is an attractive solution, but the problem does not have a straightforward solution.

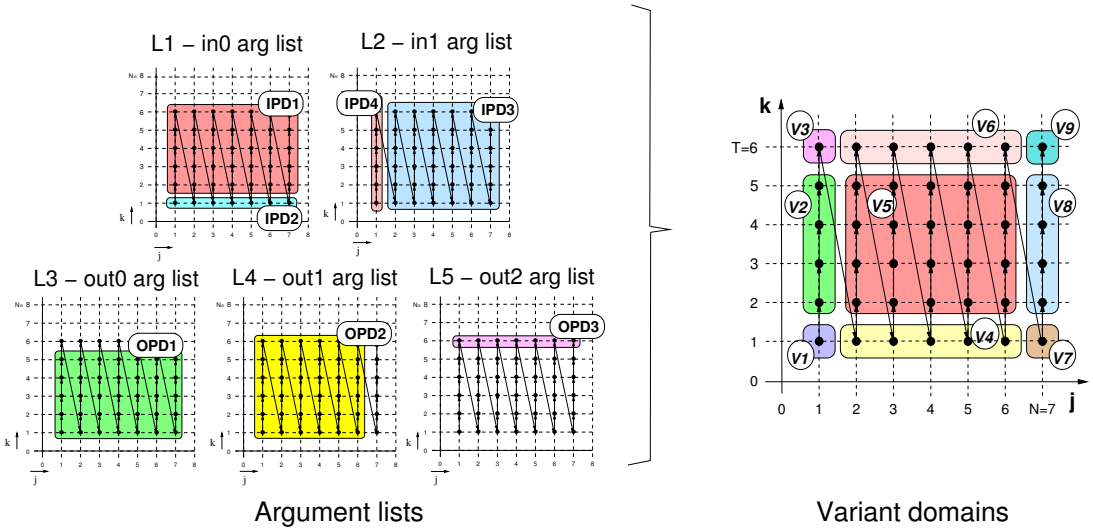


Figure 5.8.: Deriving the variant domains corresponding to a process

We proposed a partial solution to this problem in the latter part of this work. We use the fact that in Compaan, variant domains are always polyhedra (or union of polyhedra), and that all the *variants* occurring in a process can be obtained by a cross-product of all FIFO activity domains (the product operation here being a polyhedral intersection operation) as depicted in Figure 5.8. Our technique, applicable only for non parameterized domains, generates the control for a given process through the following steps:

1. build a controller that scans all variants in their order of appearance in the schedule. We use a technique based on *parametric integer programming*.
2. build a function that associates a *distance* to each transition between a variant to another one. This distance corresponds to the number of iterations in which the new variant is active. In this work, we proposed to construct this function using parametric polyhedral

counting operations based on Ehrhart [50] polynomials.

Because detailing the technical contribution is out of the scope of this overview, we will only outline the architectural model for the distance based controller (in Figure 5.7(c)).

5.4.4. Discussion

This work was done during the first months of my stay at Leiden University. Although it does not leverage sophisticated techniques, it helped significantly improve the efficiency (area, scalability) of the circuits obtained from the Compaan/Laura toolset, which was turned into a commercial product after 2005.

In retrospect, the techniques used in these papers are a bit naive and would certainly have benefited from a better background in optimizing compilers. There exist many algorithms for efficiently implementing some of the optimizations mentioned in this work (common sub-expression elimination in DAGs in particular). It also turns out that the approach based on program variants is extremely close to the technique proposed of Boulet et al. [46] that was used to address the problem of hardware control generation by Guillou et al. [75] in the same period. The algorithm of Boulet et al. is more powerful than the one sketched in this work, as it is better formalized and supports parameterized domains. Sadly, we were unaware of this work at the time.

Another missed optimization opportunity was the use of strength reduction techniques, which could have been used to reduce the complexity of many affine expression of loop indices and parameters, by using an equivalent recursive formulation. This idea was partially addressed by Zissulescu et al. [164] in 2005 and by Hannig et al. [59] in 2007 (but with a restriction to unidimensional schedules). However, we believe that there still remains a lot of work to be carried on this topic.

Summary	
	Journal: International Journal of Embedded Systems (Interscience publisher) 2008 [18]
	Workshop: Workshop on Systems, Architectures, Modeling, and Simulation 2003 [17]
	Collaborations: LIACS, Leiden University

5.5. Nested loop pipelining for HLS

The contributions presented in this section are the result of the work carried in 2011 by Antoine Morvan (supervised by Patrice Quinton). This work was carried in the context of the INRIA/STMicroelectronics Nano2012 program. In this work, we addressed the problem of *nested loop pipelining*[2], a valuable transformation for hardware synthesis, which has received only limited attention so far. We use throughout the remainder of this section a running toy loop-nest example shown in Figure 5.9, to better illustrate our contributions.

In the iteration domain of Figure 5.9, the inner loop (along the j index) exhibits no dependencies between calculations. Its execution can therefore be *pipelined* by overlapping the execution of several iterations along the j loop. *Loop pipelining* is characterized by two important parameters:

- The *Initiation Interval* (denoted II in the following), that corresponds to the number of clock cycles separating the execution of two loop iterations.
- The *latency* (denoted Δ) that gives the number of clock cycles required to completely execute one iteration of the loop.

The rightmost part of Figure 5.9 depicts the pipelined execution of our example, with an initiation interval II=1 and a latency of $\Delta = 4$. Loop pipelining is a key transformation in

2. Not to be confused with multidimensional pipelining or outer loop pipelining

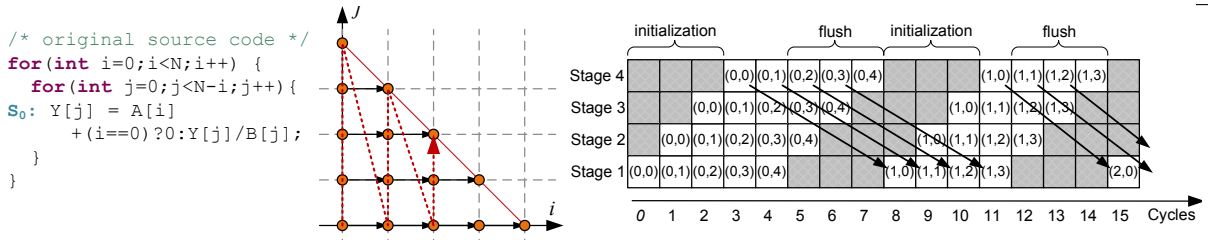


Figure 5.9.: Motivating example, with its pipelined execution for $N = 5$, II=1 and $\Delta = 4$.

High-Level Synthesis tools, as it helps maximizing the computation throughput and because it improves hardware utilization. Thus, fully pipelining the loop (that is choosing II=1) is a very common practice. For loops with large iteration count this leads to almost 100% hardware utilization.

However, when the iteration count of the loop is comparable to the pipeline latency Δ, one may observe a significant performance degradation, as the pipeline flushing phases dominate the execution time. This is the case of our example in Figure 5.9. For a value of $N = 5$ and $\Delta = 4$, we obtain a hardware utilization rate of only 50%. This schedule is far from being efficient, and the reader can see that there is not reason to wait until $t = 8$ to start the second iteration of the outer loop, and that the whole loop nest execution might be pipelined. Such an optimization is known as *nested loop pipelining* [114], and can be realized through a *loop coalescing* transformation that flattens a loop nest into a single loop that scans the original iteration domain. *Nested loop pipelining* can then be done by pipelining this coalesced loop. Implementing nested loop pipelining (and in particular enforcing its correctness) is far from trivial and requires a lot of attention.

As an example, Figure 5.10 shows a coalesced version of the loop nest of Figure 5.9. One can observe that the array accesses in the coalesced version do not depend on loop indices as in Figure 5.9. They are hence more difficult to analyze. As HLS tools rely on simple data-dependency analysis algorithms, they will fail to detect that this loop can be pipelined. They hence offer compiler directives (i.e., `#pragma`) that can be used to force the tool to ignore user-specified memory references in its dependency analysis, to enable pipelining. Of course, this comes with the risk of generating illegal pipelined schedules.

In our example, we may be tempted to bypass some of the dependence analysis through a `#pragma ignore_mem_dep Y` directive to ignore the dependency over `Y[j]` and enable the whole loop pipelining.

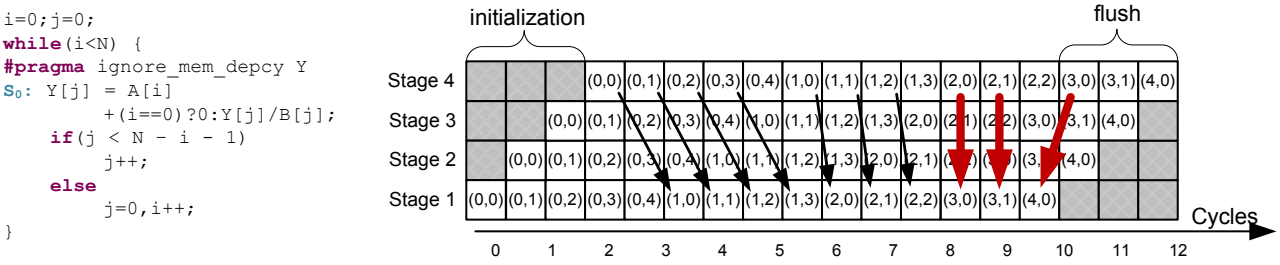


Figure 5.10.: Illegal nested loop pipelining for $N = 5$, $II = 1$ and $\Delta = 4$. Bold arrows show broken dependencies.

Although the scheduling seems correct at the first glance, that some Read after Write dependencies are violated for $i \geq 3$, as shown in Figure 5.10. For example, the memory read operation on `Y[0]` of $(i = 3, j = 0)$ scheduled at $t = 12$ happens before `Y[0]` is updated by the write operation of $(i = 2, j = 0)$, also scheduled at $t = 12$ on the last stage.

Among the numerous commercial and academic C to hardware tools that we have evaluated, only Catapult-C from Mentor Graphics actually provides the ability to perform such automatic nested loop pipelining. However, its current implementation in the tool suffers from severe flaws and generates illegal schedules whenever the domain is not rectangular and/or has non constant loop bounds (from what we understand the tool fails for the very same reasons as depicted in Figure 5.10).

Our contribution in this work is threefold. We provide a formalization of the conditions under which nested loop pipelining is legal with respect to data dependencies for SCoPs programs [40]. In addition to this legality check, we also propose a technique to *correct* a priori illegal nested pipeline schedules by inserting *bubbles* in the coalesced loop, to derive the most efficient *legal* pipelined schedule. Finally, we also propose an implementation of our *polyhedral bubble insertion* technique in a source-to-source compiler, to remain as vendor independent as possible.

5.5.1. Checking pipelined coalescing legality

In the following, we outline the idea behind our nested pipelining legality check. For a more detailed technical description of our solution, we incite the reader to refer to the article by Morvan et al. [22] provided [3] in appendix B. Consider a sink reference to an array (i.e., a read array reference in a statement), and let $\vec{y}$ denote its iteration vector. Let $\vec{x}$ be the iteration vector of the source of this array reference. Let us define d to be a function that maps $\vec{y}$ to $\vec{x}$, such that $\vec{x} = d(\vec{y})$. We can formulate the lagality conditions of loop coalescing as follows: for a nested pipeline schedule with a latency of Δ and $II = 1$, the schedule violates data dependencies if the reuse distance (in number of iteration points) between the production of the value (at iteration $\vec{x}$, the source) and its use (at iteration $\vec{y}$, the sink) is less than Δ.

We build a function $next_{\mathcal{D}}^{\Delta}(\vec{x})$ that computes, given an iteration vector $\vec{x}$, its successor Δ iterations away in the loop nest iteration domain. To test this condition, we then check that all the sink iteration vectors $\vec{y} = d^{-1}(\vec{x})$ of the dependency d occur after $next_{\mathcal{D}}^{\Delta}(\vec{x})$. We derive the $next_{\mathcal{D}}^{\Delta}$ function by using earlier results by Boulet et al. [46] who provide a technique to compute the immediate successor in $\mathcal{D}$ of an iteration vector $\vec{x}$ according to the lexicographical order. Because we only need to look for a constant number of iterations ahead (the latency of the pipeline that we call Δ), we can easily build the $next_{\mathcal{D}}^{\Delta}$ function. This is done by applying the function to itself Δ times. Let us compute the $next_{\mathcal{D}}(\vec{x})$ function for the example of Figure 5.9, with $\vec{x} = (i, j)$. We have

$$next_{\mathcal{D}}(i,j) = \begin{cases} (i, j+1) & \text{if} & j < N - i - 1 \\ (i+1, 0) & \text{elseif} & i < N - 1 \\ \bot & \text{otherwise} & \end{cases}$$

Note that $\bot$ represents the absence of successor in the loop. Applying the function four times

3. Because it was not yet presented at the conference at the time of this writing, we provide a draft of the camera ready version in this document

to itself gives the $next^4_{\mathcal{D}}(i,j)$ function:

$$next^4_{\mathcal{D}}(i,j) = \begin{cases} (i,j+4) & \text{if} & j \leq N-i-5 \\ (i+1,3) & \text{elseif} & i \leq N-5 \wedge j = N-i-1 \\ (i+1,2) & \text{elseif} & i \leq N-4 \wedge j = N-i-2 \\ (i+1,1) & \text{elseif} & i \leq N-3 \wedge j = N-i-3 \\ (i+1,0) & \text{elseif} & i \leq N-4 \wedge j = N-i-4 \\ (N-1,0) & \text{elseif} & i = N-3 \wedge j = 1 \wedge N \geq 3 \\ (N-2,0) & \text{elseif} & i = N-4 \wedge j = 3 \wedge N \geq 4 \\ \perp & \text{otherwise} & \end{cases} \tag{5.1}$$

An interesting property of this approach is that it allows us to determine the domain of the iterations causing a data dependency violation. In the following, we call this domain $\mathcal{D}^\dagger$. This domain can be automatically derived using simple polyhedral operations, and whenever this domain is empty, nested pipelining is legal. In our example, we obtain:

$$\mathcal{D}^\dagger = \{i,j | (i,j) \in \mathcal{D} \wedge N-4 < i < N-1 \wedge j < N-i-1\}$$

When we substitute N by 5 (the chosen value in our example), we have $\mathcal{D}^\dagger = \{(2,0),(2,1),(3,0)\}$, which is the set of points that causes a dependency violation in Figure 5.10.

5.5.2. Correcting coalescings by bubble insertion

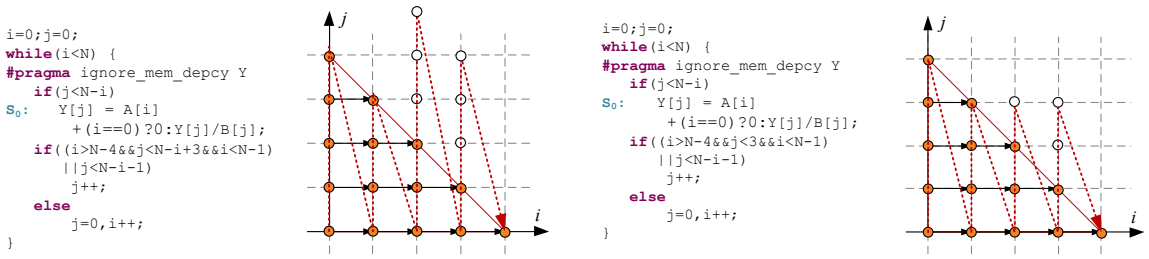
While a legality condition is an important step toward automated nested loop pipelining, we would also like to *correct* a given schedule to make the coalescing legal. This is achieved by inserting *at compile time* dummy iterations (or *bubbles*) in the iteration domain, that delays the execution of the iterations causing dependency violations. The key question in this problem is to determine how many of such wait states are actually required to fix the schedule, as adding a single wait state in a loop may incidentally fix/correct several violated data-dependencies. We have proposed two different approach to the problem, illustrated in Figure 5.11(a) and Figure 5.11(b)

The simplest solution is to pad every inner loop containing an iteration in $\mathcal{D}^\dagger$ with $\Delta-1$ wait-states (this boils down to flushing the pipeline only when needed). The approach is illustrated in Figure 5.11(a), but turns out to be too conservative. For example, the inner loops for indices $i = 2$ in the example of Figure 5.9 do not need $\Delta - 1 = 3$ additional cycles. In this case, only one cycle of wait state is needed, and similarly, for $i = 3$, only two cycles are needed.

An alternative approach, illustrated in Figure 5.11(a), is to build the set of additional wait states by first looking to the source iterations in $\mathcal{D}^\dagger$ that cause a schedule violation by exactly 1 cycles and then pad the inner loop corresponding to these iterations with exactly 1 wait-state. We then focus on the source iterations that fail by exactly 2 cycles and pad the inner loop corresponding to these iterations with exactly 2 wait states, possibly overlapping some of the wait states previously introduced. The algorithms then proceeds until it has inserted wait states for the iterations that fails by $\Delta - 1$ cycle, i.e., until all schedule violations are resolved.

5.5.3. Results and validation

In this section, we describe how the transformation is implemented within a compiler framework. We also provide quantitative evidence showing that the approach is practical and leads to performance improvements at the price of a moderate increase in area.



(a) Conservative pipeline correction

(b) Optimized pipeline correction

Figure 5.11.: Two examples of pipeline correction through bubble insertion

Implementing the loop coalescing transformation

Implementing the loop coalescing transformation amounts to a rewriting of the loop nest structure into a software finite state machine expressed as a loop. There are two possible approaches for implementing this rewriting.

The first approach uses the Control Flow Graph (CFG) corresponding to the loop nest as an input. The problem with this approach is that the automaton is built from an implicit representation of the iteration domain rather than from its formal representation as a polyhedron. As a consequence, the automaton contains extra idle states that do not correspond to an actual iteration of the loop nest. From what we understand, this is the approach followed by Catapult-C when implementing the *nested loop pipelining* transformation.

The second approach follows the approach by Boulet et al. [46] and consists in building a finite state machine directly out of the loop nest iteration domain. With this technique, the generated code visits the *exact* loop nest iteration domain, leading to a more efficient approach. However, the resulting code tends to be more complex in term of guards and induces some area overhead.

Because one generally does not want to coalesce the full loop nest, our approach allows the designer to choose (by using a compiler directive) how many of the innermost loops should be coalesced.

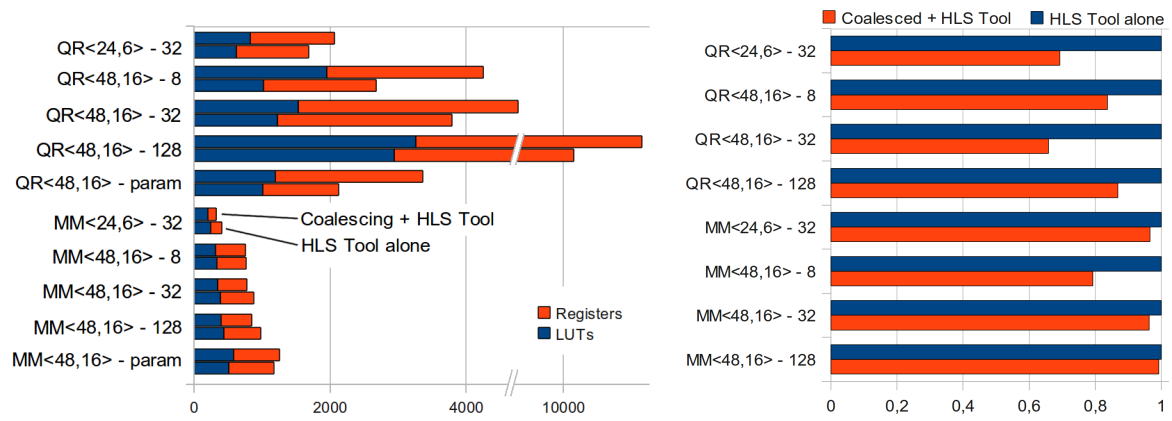
Experimental results

The efficiency of our approach is sensitive to the trip count of innermost loops. It is therefore more sensitive to the iteration domains size and domain shapes than to the structure of the application itself. In this work, we limited ourself to two representative kernels (QR factorization and matrix multiplication) that provide a good illustration of the benefits and drawbacks of the approach.

For each of these kernels, we used varying fixed and parameterized iteration counts. We also used different fixed point arithmetic word length sizes for the algorithm, to be able to precisely quantify the trade-off between performance improvement and area overhead. The kernels underwent aggressive pipelining (with II=1 and Δ varying from 4 to 12).

The quantitative evaluation of the area cost induced by the use of nested pipelining is provided in Figure 5.12(a). Our results show that the area overhead remain limited when larger functional units are being used. Also, our approach does not significantly impact the clock frequency (less than 5% difference in all cases).

The improvement in execution time due to latency hiding are given in Figure 5.12(b), it can be seen that for larger iteration counts, the performance improvements are limited and



(a) Area cost

(b) Normalized execution time

Figure 5.12.: Experimental results (area and speed) for our nested pipelining approach. The label `<a,b>` stands for a `a` bit wide fixed point format with `b` bit for integer part)

barely justify the area overhead. When no correction is needed (e.g., constant size matrix multiplication) our coalescing transformation is more efficient in performance and area than the nested pipeline feature of Catapult-C. In addition, for the *QR* kernel, any attempt to use this feature leads to an illegal schedule if the iteration domain is parameterized. Thus, our experiments provide results for a non-nested pipelined version of the QR kernel.

5.5.4. Related work

Loop pipelining is a well studied problem, with numerous contributions since the seminal work of Lam in 1988 [97]. In the following, we describe some closely related work.

Rong et al. [134] have already studied the problem of nested loop software pipelining. Their goal is clearly the same as ours, except that they restrict themselves to a narrow subset of loops (only constant bounds, rectangular domains) and do not leverage exact instance-wise dependence information. They also do not address the problem from a hardware synthesis point of view. In this work, we tackle the problem for a wider class of programs (known as Static Control Parts), and we also relate the problem to *loop coalescing*.

Another related contribution is the work of Fellahi et al [66]. They address the problem of prologue/epilogue merging in sequences of software pipelined loops. Their work is also motivated by the fact that the software pipelining overhead tends to be a severe limitation as many embedded-multimedia algorithms exhibit *low trip count* loops. Again, our approach differs from theirs in the scope of its applicability, as we are able to deal with loop nests (not only sequences of loops), and as we solve the problem in the context of HLS tools at the source level through a loop coalescing transformation. Their approach handles the problem at machine code level, which is not possible in our context.

A recent work by Alias et al. [34] tackles a problem similar to ours, as they try to address the problem of generating efficient nested loop pipelined hardware accelerators leveraging custom floating point datapaths. Their approach (also based on the polyhedral model) consists in finding a parallel hyperplane for the loop nest, and then deriving a tiling (hyperplanes and tile sizes) such that a pipeline of depth Δ is legal. The approach only targets perfectly nested loops and also requires incomplete tiles to be padded to behave like full-tiles. They also restrict themselves to uniform dependencies, to guarantee that the reuse distance (i.e., the number of

points separating a source and a sink) is always constant for a given tile size. In contrast, our framework is more general and supports imperfectly nested loops with affine dependencies. In addition, in the case of tiled iteration domains, we can provide a more precise correction that does not require the padding of all incomplete tiles.

5.5.5. Discussion

The results presented in this section correspond to an on-going work, and we are currently working on extending the applicability and relevance on the approach.

The improvements offered by the approach may not be obvious, as the area overhead caused by the extra *bubbles* control may not always compensate for the improvements of performance. Nevertheless, the ability to prove the legality of the nested pipeline transformation remains a valuable tool. Besides, as mentioned in Subsection 5.4.4 we believe that there are still many unexplored opportunities for reducing this area overhead by improving the quality of the generated code. Another interesting consequence of nested pipelining is that it allows designers to consider using deeper pipelines, a strategy they are often reluctant to follow because of the possibly large fill/flush overhead.

We are also planning to evaluate our approach on vectorized pipelined accelerators. In these accelerators, parallelism is also expressed at a coarser grain, by using distinct datapaths for several iterations of the parallel innermost loop. In such a scenario, our approach is likely to be even more effective, for the two reasons given below.

- Because of this parallel execution, the number of iterations within this pipelined and vectorized innermost loop will be smaller, increasing the (negative) impact of the pipeline latency on the overall performance.
- Because the accelerator will then instantiate several datapath (one for each parallel iteration), the impact of the control logic on the overall area budget will become almost negligible.

However, the ability to derive vectorized and pipelined accelerators poses additional challenges (especially in terms of conflicting memory accesses) that need to be addressed before we can extend our approach. We are currently investigating this topic.

Chapter 6

Conclusion

As explained in the first chapter, the work presented in this document addresses several closely related topics: the proposal of original hardware platforms (Chapters 2 and 4), the evaluation of design methodologies through cases studies (Chapters 2 and 3), and several contributions to design automation tools (Chapters 4 and 5). The reader may have noticed that most of my recent contributions focus on design tools. Indeed, my research interests since 2003 have been slowly shifting away from high performance reconfigurable platforms to CAD tools and parallelizing compilers.

This evolution was made possible thanks to the work of Ludovic L'Hours on the Gecos compiler, which allowed me to benefit from a solid software infrastructure which we have been able to extend to carry several new research projects. Gecos now serves as a prototyping framework for many CAIRN members and enabled new research activities within the group. Another enabling factor was the start of a collaboration with the Triskell group at IRISA, which studies and researches Model Driven Software Design practices.

In the remaining of this chapter, I will sketch my shorter and longer term research perspectives, which focus on three topics:

- **Improving the efficiency of High-Level Synthesis tools**, by tackling HLS compiler specific problems directly at the front-end level, through source-to-source transformations. In particular, I believe that it is time to revisit hardware synthesis techniques based on polyhedral representations of programs, by taking advantage of the numerous breakthroughs occurred in the last few years. I also plan to study how formal static analysis techniques could be used to improve the efficiency of HLS tools.
- **Efficient parallelization on heterogeneous embedded multi-cores**. I plan to study how constraint-based programming approaches can help automatic parallelization in the polyhedral model for heterogeneous platforms, with a focus on runtime adaptivity. I would also like to pursue the work started in collaboration with Colorado State University on the design of programming tools to help software developers designing correct and efficient parallel programs.
- **Model Driven Engineering and optimizing compilers**. This research direction is very recent and is the result of a new collaboration with the Triskell Group at IRISA. The main goal of this of this research direction is to contribute to the cross-fertilization of Model Driven Engineering and optimizing compiler design techniques and in particular to propose techniques to ease the design of domain specific optimizing compilers, by enabling the reuse of complex analyses and transformations over families of languages.

6.1. Toward next generation of High-Level Synthesis tools

While High-Level-Synthesis has made huge progresses over the last decade, most tools are still very far from being actual C to hardware compilers, and should rather be seen as micro-architectural specification languages based on C/C++. In these tools, the designer implicitly exposes the target micro-architecture in its source code and has to rely on complex annotations to help the tool infering an efficient circuit out of this specification.

There is therefore still a long way to go to make HLS tools more widely usable and more

efficient when it comes to deriving customized circuits. The following topics address parts of this problem and aim at contributing to the design of next generation HLS tools.

6.1.1. Revisiting hardware synthesis in the polyhedral model

As mentioned in Section 5.1, the polyhedral model has experienced several breakthroughs over the last 10 years, which significantly improved its efficiency and applicability. Most of these breakthroughs focused on improving the performance of programs on parallel programmable machines (multi-cores and GPUs) with a shared memory parallel programming model. In particular the Pluto compiler [44], proposed in 2008, set a new landmark in terms of automatic parallelization, as it was the first to address parallelism and data reuse within the same framework.

Given these achievements, we believe that there is a need for revisiting the way we have been tackling hardware synthesis in the polyhedral model. In particular, it seems to us that source-to-source transformations used as front-end of HLS tools, is a very interesting approach. It first avoids spending too much effort on the hardware back-end, leaving it to the HLS tools and permits to remain vendor neutral. It also makes the contribution more widely usable by the HLS research and user community.

Very interesting preliminary results have been obtained for communication optimizations by Plesco et al. [123], but many other topics remain to be addressed before the power of the polyhedral based transformations can be fully utilized by HLS tools. The contribution presented in Section 5.5 is a first step in this direction, and we plan to continue exploring the above mentioned problems during the third PhD year of Antoine Morvan.

6.1.2. Domain Specific Analyses for HLS

Interestingly, the nested pipelining presented in Section 5.5 is an optimization that bears little interest in the context of a standard optimizing compiler, but has significant added value in the scope of a hardware synthesis tool. We believe that there exists many other analyses/optimizations that fall into this category, and HLS tools could benefit a lot from more domain specific compiler optimizations, as they could leverage their results to generate faster and/or smaller circuits.

Because compilation runtime is not so critical for HLS as compared to a standard compiler (compile times in the order of several minutes for a small program is common), very aggressive analyses and optimizations could be considered. Such analyses could, among other possibilities, take advantage of the large choice of open, and high quality formal verification tools (Frama-C [31], InterProc [142], Aspic [72]) and libraries (SMT solvers such as Yices [58]), which are currently used for verification purposes, but whose results may prove to be very valuable to HLS.

A typical example is a bit-level wordlength analysis of the expressions and variables of a program. This information can be used in the scheduling/binding process to reduce the hardware footprint. While simple algorithms exist to estimate such bitwidth, they do suffer from very limited accuracy, and it would certainly be worth studying how more advanced formal techniques (such as those based on abstract interpretation) could help addressing such problems.

In spite of its importance, this niche remains currently unexplored, this for simple reasons. Most researchers in HLS focuses on back-end issues, they hence tend to ignore higher level optimizations and analyses, and tend to consider that “this is the front-end compiler job”. On the other hand, compiler research is faced with so many challenges that very few researchers are interested in such niche problems. As a consequence, they tend to consider on their side that “this is a hardware synthesis issue, not a compiler issue”.

We believe that such problems make a very attractive research topic, with many interesting and open research problems. This is a research topic that I clearly plan to continue investigating in both short and long term.

6.2. Automatic parallelization for heterogeneous multi-cores

As mentioned in the introduction, embedded system designers are also now confronted to the outbreak of multi and many-core architectures. Efficiently designing and programming such systems poses significant challenges, mostly because of the heterogeneity of these platforms. In the following, we sketch three directions that we would like to pursue in the future.

6.2.1. Constraint programming for automatic parallelization

Over the last few years, *iterative compilation* techniques have become very popular for general purpose computing, as it has been acknowledged that the subtle interplay between compiler optimizations and modern general purpose processors micro-architectures has become too complex to be captured by (even very complex) explicit performance models. For example, the work of Pouchet et al. [124] studies the applicability of iterative compilation to the polyhedral model, by using a meta heuristic to explore the space of legal transformations for a loop nest. The goal is to find the best transformations for a given architecture/compiler combination, with a direct performance measurement for each of the transformed program (thousands of program instances can be explored).

While being very powerful, such an approach is not possible in the context of embedded platforms, since the hardware is generally designed jointly with the software. Performance evaluation is then performed through complex platform level simulators [1], whose limited speed prevents the exploration of thousands of program instances. On the other hand, because the behavior of these platforms is generally more predictable than that of general purpose machine, the use of partial explicit performance models remains relevant. The problem is then to be able to use those performance models to efficiently guide the design space exploration, so as to pick up the most profitable transformations.

The work of Prof. Wolinski (PhD of Kevin Martin and Antoine Floc'h), which studied the application of constraint based programming for solving compilers related combinatorial optimization problems, has clearly demonstrated the relevance of this technique for this type of problems. In this context, we would therefore like to study how it is possible to use these techniques to solve the numerous non convex optimization problems that arise in a polyhedral based exploration approach, in particular when fine grain resource constraints are involved.

6.2.2. Adaptive run-time parallelization for heterogeneous multi-core

Loop Tiling (and more specifically the choice of the tile size) is an effective (and well known) technique for coarse-grain parallelization. The choice of the tile dimensions is often the result of subtle trade-off between the amount of parallelism (decreasing as tile size grows) in the tiled program and the overhead of communications and cache-misses (also decreasing as tile size grows). Recent work on Parameterized Tiling code generators [129, 38] open interesting perspectives, as they do not impose anymore the choice of tile size at compile time. In particular some authors have advocated to change the tile size at runtime [148], directly during the tiled loop nest execution, to be able to adaptively optimize the program performance by monitoring

1. Based on Instruction Set Simulators for programmable processors to cycle accurate models for custom co-processors.

the execution of the program trough performance counters. We believe this is a promising research direction, and plan to explore this technique in the context of adaptive heterogeneous multiprocessors where computing and storage resource may evolve in time.

6.2.3. Parallel programming tools for non parallel programmers

Software designers do nowadays benefit from highly sophisticated Integrated Design Environments that provide very valuable tools (e.g. refactorings, static analysis), which helps them reducing development time. Because of the generalization of multicore, these users will sooner or later be exposed to some form of parallel programming, and will hence need similar tools for parallel programs. The role of such tools will be to help the programmer finding functional or performance bugs [2] in their parallel implementation. Following the idea developed in subsection 6.1.2, we believe that there are many cases where advanced static analysis tools can be used to that purpose.

In the context of a collaboration with Colorado State University, we already started addressing the problem, and have designed a static analysis tool called ompVerify [2]. This tool, integrated within the Eclipse IDE, checks for the correctness of openMP programs, by looking for possible data races in the program. Although the analysis, based on very well known results in the domain of loop parallelization, is quite straight forward, it was perceived as being very useful by many openMP programmers. Interestingly, and despite the large openMP users community, such a tool did not exist [3] so far. We are currently considering extending the applicability of the analysis by using Satisfiability Modulo Theory solvers, which could permit to handle a much wider class of programs, in particular those using non affine array accesses.

6.3. Model Driven Engineering and optimizing compilers

This new research topic is the consequence of a recent collaboration with the Triskell research group at IRISA, which focuses on software engineering issues and on Model Driven Engineering and software design. MDE offers tools and facilities to ease the definition, analysis and tooling of Domain Specific Modeling languages (by languages, we mean either a textual or graphical notation, following well defined syntax and semantics). The techniques turned out to be also very useful in the context of optimizing compiler infrastructures. We have been extensively using MDE within the Gecos compiler framework since 2008, which helped us a lot in trying to prototyping new compilers and CAD flows. This work has also led to a publication at the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems [19] in 2011.

In particular, facilities such as Xtext, which allow programmers to create new Domain Specific Languages and their corresponding IDEs in a matter of days open very exciting perspectives from a Compiler/CAD point of view, that we discuss below.

6.3.1. Domain Specific Languages for high productivity parallel computing

Programming complex parallel system in an efficient way will remain out of reach for most developers. A more systematic use of highly Domain Specific Languages, tailored to match the expertise domain of the user, can be an (partial) answer to the problem.

In such a specialized language, it becomes possible for its associated compiler to take advantage of the domain specific semantics of the DSL, for discovering and/or taking advantage

2. By performance bug, we mean an ill designed programmed with suffer from very low performance.
3. Syntactic static analyses do exist but are unable to handle array accesses for example.

of potential source of parallelism and data reuse that would otherwise have been missed by a compiler operating on a general purpose programming language.

We have started investigating this topic in the context of the Grappas Project, in collaboration with the IETR research laboratory. Our goal is to see how a DSL approach could be used to help antenna designers to generate efficient parallel implementations for GPUs and multicore of numerical simulation codes based on the FDTD algorithm, a stencil based computational pattern.

6.3.2. Software reuse in MDE through model typing

At first glance, the ability to create new languages is a very attractive solution, it however suffers from a significant weakness: development effort. As every language will need its own optimizing compiler toolchain, this makes the approach quite unrealistic. However, this limitation could be partially circumvented if it was possible to reuse common compiler optimizations and analyses over all these languages (or at least among families of languages). This is of course a challenging problem, as in addition to the semantics variations occurring among languages, their intermediate representations can also be very different. This means that, even if the target transformation is relevant (from a semantic point of view) with respect to a given language, reusing the algorithm *as is* is generally not possible. In the context of the PhD topic of Clément Guy (supervised by J.M. Jezequel), we are currently looking at extending the notions of Model Typing [143], which captures the substitutability relation among languages, in order to address this reusability issue.

Appendix A

Curiculum Vitae

Personnal Informations

Steven Derrien
29 Rue Durafour
35000 Rennes
37 ans.

Contact

ISTIC/IRISA
Campus de Beaulieu
35000 Cedex Rennes
sderrien@irisa.fr

Current Position

Since sept. 2009 **Associate professor** at Université de Rennes 1, in **INRIA sabbatical**

Previous positions

Since sept. 2003 **Associate professor** in Computer Science, Université de Rennes 1

September 2001 to august 2003 **Assistant professor** in Computer Science, Université de Rennes 1
ATER at IFSIC

October 1998 to december 2002 **PhD student at Université de Rennes 1**, within the COSI research group at IRISA.

March to August 1998 **Master internship** at the Stanford Research Institute (Palo Alto, USA).
Topic: stereovision using panoramic imaging devices for mobile robots.

Academic background

1998 - 2002 **PhD in Computer Science from Université de Rennes 1** under the supervision of Sanjay Rajopadhye. Title: **Etude quantitative des techniques de partitionnement de réseaux de processeurs pour circuits FPGA.**

Defended December 2nd, 2002. Jury composition Anne Mignotte (reviewer), Eric Martin (reviewer), Patrice Quinton (Jury Chair), Olivier Sentieys, François Charot

1997 - 1998 **DEA Signal Telecommunication Image Radar** University of Rennes I

1995 - 1998 **Diplôme d'ingénieur ENSSAT Lannion**, Computer Engineering Major.

Awards

2007 **Best paper award** at 18th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2007)

Teaching activities

I have been teaching within the Computer Science Department [1] of University of Rennes 1 since 2003. Before that period, I also had the chance to work as a teaching assistant (Moniteur) and as assistant professor (ATER).
Since I was hired as associate professor in 2003, I have mostly been teaching courses in computer architecture with a focus on embeded systems and in operating system design (hardware/software interface). More recently, I started participating to "software projects" at the master level.
I also participated to some introductory courses to computer programming (in Java) for second year students, many of whom study computer science as a minor.
A summary of my teaching activities (along with the corresponding number of teaching hours) is given below, in particular Courses (Courses/CM, Practice/TD, Labs/TP, Projects) which I completely supervised and/or that I created or significantly modified are shown in bold font.

- *L2 Informatique*
  - API: Imperative programming in Java (2003-2009)(**TD: 24h, TP: 48h**)

- *L3 Informatique*
  - **SYR1: Operating Systems 1 (2003-2009)**(**TD: 28h, TP: 24h**)
  - **SYR2: Operating Systems 2 (2003-2009)**(**TD: 38h, TP: 32h**)

- *M1 Informatique*
  - **Hardware architectures for Embedded Systems (2009)(CM: 24 h, TD: 12 h, TP: 12 h)**
  - **Software development project (PROJ) (2008-2010)(TD: 20h, TP: 20h)**

- *Engineering degree DIIC-ARC, 2nd year (2003-2009)*
  - **Embedded System Design (2009) (module CSE, 2003-2009)** (**TP: 48h**)

- *Engineering degree DIIC-ARC, 3rd year (2003-2009)*
  - **Final year student project (2003-2008)**(**TP: 24h**)

- *Master in Computer Science (components and software systems)*
  - **Hardware System Synthesis (CSS)**(**CM: 4h**)

1. which used to be IFSIC and is ISTIC

Projects and grants

International projects

- Head of the LRS INRIA international research team with the Mélange group of professor Sanjay Rajopadhye at Colorado State University since 1/1/2010. The LRS research group focuses on automatic parallelisation techniques and tool for heterogeneous embeded multi-core architectures and GPUs.
- Participant to the FP7 ALMA project (starting in October 2011). The goal of the project is to propose a complete design flow for embedded heterogeneous multi-core architectures from a Matlab-like language.

National projects

- French National Research Agency (ANR) project: COMPA (Model of Computation Driven Design for Adaptive Multi Processors), starting in 2011. One PhD funded for a topic in collaboration with M. Raulet from INSA Rennes.
- S2S4HLS project (2009-2012), Source-to-Source transformations for High-Level Synthesis (1 PhD grant + 2 Yrs Engineers). The project is a collaborative project between INRIA and STMicroelectronics in the context of the Nano 2012 funding.
- French National Research Agency (ANR) project: BioWic (Bioinformatics Workflows for Intensive Computation), started in 2009, ending in 2012. One PhD student funded (Naeem Abbas).
- French National Research Agency (ANR) project Sémim@ges (2007-2009) one year post-doc funded
- ACI ReMIX (2003-2006), Reconfigurable Memory for Indexing Huge Amount of Data

Community service:

- Reviewer for the following conferences and journals: DAC, DATE, FPL, ASAP, ERSA, IEEE Transaction on VLSI, IEEE Design and test of computers, ACM Transaction of reconfigurable computing and Transaction on Embeded systems.
- Publicity chair for the 2010 ASAP conference
- Program Committee member for the French Sympa 2011 conference
- Program Committee member for the Architecture of Computing Systems International conference in 2009.

Supervision

PhD Students & post-docs

- Clément Guy, PhD Student (under the supervision of Jean Marc Jezequel) and in close collaboration with Benoit Combemale since 1/10/2010. PhD topic: Generic Definition of Domain Specific Analysis using Model-Driven Engineering (MDE). Involvement 20%, defense scheduled by the end of 2013.
- Vivek Tomare, PhD Student (under the supervision of Olivier Sentieys) since le 1/10/2009, PhD topic: Reconfigurable Low Power Wireless Sensor Network Nodes. Involvement 20%, defense scheduled by the end of 2012.
- Antoine Morvan, PhD Student (under the supervision of Patrice Quinton) since le 1/10/2009, PhD topic: Source-to-source transformations for High Level Synthesis. funded by INRIA/-Nano2012. Involvement 80%, defense scheduled by the end of 2012
- Naeem Abbas, PhD Student (under the supervision of Patrice Quinton) since 5/2/2009, PhD topic: Reconfigurable architectures for accelerating bioinformatic Workflows. Funded by an ANR Grant (BioWic). Involvement 80%, defense scheduled first half of 2012.
- Adeel Pasha, PhD Student (under the supervision of Olivier Sentieys), PhD topic: Ultra-Low Power Controllers for Wireless Sensor Network Nodes, MESR funding, defended December 15th 2010. Involvement 70%.
- Auguste Noumsi, PhD Student (under the supervision of Patrice Quinton), PhD topic: Architectures for multimedia content retrieval, defended November 10th 2010. Involvement 30%.
- Florent Berthelot, post-doct from 1/10/2007 to 31/8/2008, funded by the ANR Semim@ge project. Involvement 80%.

Research engineers

- Amit Kumar, from 1/11/09 to 1/11/11 funded by the S2S4HLS/Nano2012. Involvement 100%.
- Maxime Naullet, from 15/10/10 till 15/10/12 in the context of the Kergecoz INRIA ADT, which focuses on using MDE to help building compiler infrastructures. Involvement 90%.

Masters student

- Youcef Barigou, from 1/2/11 to 31/6/11, topic: Parallelization of real life stencil codes on GPU architectures. Involvement 50%.
- Nina Engelhardt, stagiaire M2R du 1/2/10 au 31/6/10, topic: Multi-mode synthesis applied to the automatic generation of programmable processor datapaths. Involvement 100%.
- Clément Guy, from 1/2/10 to 31/6/10, topic: Extending the datapath merging algorithm to regular architectures. Involvement 100%.
- Antoine Morvan, from 2/09 to 06/09, topic: Synthesis of hardware controllers for scanning loop nests. Involvement 100%.
- Jean-Baka Domelevo, from 2/05 to 06/05, topic: Energy optimization in nested loop accelerators using bit-level data correlation. Involvement 100%.

Appendix B

Selected publications

In this section we provide the camera ready version of the article describing the work detailed in Section 5.5 that will be presented at the International Conference on Field-Programmable Technology (FPT'11) on December the 14^{th}, 2011. This appendix will be removed from the final manuscript and is provided for the sake of completeness.

authors	Antoine Morvan and Steven Derrien and Patrice Quinton
title	*Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion*
conference	*International Conference on Field-Programmable Technology (FPT'11)*
date	December 2011
pages	proceedings not yet available

Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion

Antoine Morvan [1], Steven Derrien [2], Patrice Quinton [1]

[1] *INRIA-IRISA-ENS Cachan*
[2] *INRIA-IRISA-Université de Rennes 1*
Campus de Beaulieu, Rennes, France
{amorvan,sderrien,quinton}@irisa.fr

***Abstract*—Loop pipelining is a key transformation in high-level synthesis tools as it helps maximizing both computational throughput and hardware utilization. Nevertheless, it somewhat looses its efficiency when dealing with small trip-count inner loops, as the pipeline latency overhead quickly limits its efficiency. Even if it is possible to overcome this limitation by pipelining the execution of a whole loop nest, the applicability of nested loop pipelining has so far been limited to a very narrow subset of loops, namely perfectly nested loops with constant bounds. In this work we propose to extend the applicability of nested-loop pipelining to imperfectly nested loops with affine dependencies by leveraging on the so-called polyhedral model. We show how such loop nest can be analyzed, and under certain conditions, how one can modify the source code in order to allow nested loop pipeline to be applied using a method called *polyhedral bubble insertion*. We also discuss the implementation of our method in a source-to-source compiler specifically targeted at High-Level Synthesis tools.**

I. Introduction

After almost two decades of research effort, High-Level Synthesis (HLS) is now about to hold its promises : there now exists a large choice of robust and mature C to hardware tools [1], [2] that are even now used as production tools by world-class chip vendor companies. However, there is still room for improvement, as these tools are far from producing designs with performance comparable to those of expert designers. The reason of this difference lies in the difficulty, for automatic tools, to discover information that may have been lost during the compilation process. We believe that this difficulty can be overcome by tackling the problem directly at the source level, using source-to-source optimizing compilers.

Indeed, even though C to hardware tools dramatically slash design time, their ability to generate efficient accelerators is still limited, and they rely on the designer to expose parallelism and to use appropriate data layout in the source program.

In this paper, our aim is to improve the applicability (and efficiency) of nested loop pipelining (also known as nested software pipeling) in C to hardware tools. Our contributions are described below:

- We propose to solve the problem of nested loop pipelining at the source level using an automatic *loop coalescing* transformation.
- We provide a nested loop pipelining legality check, which indicates (given the pipeline latency) whether the pipelining enforces data-dependencies.
- When this condition is not satisfied, we propose a correction mechanism which consists in adding, at compile time, so-called *wait-states* instructions, also known as *pipeline bubbles*, to make sure that the aforementioned pipelining becomes legal.

The proposed approach was validated experimentally on a set of representative applications for which we studied the trade-off between performance improvements (thanks to full nested loop pipelining) and area overhead (induced by additional guards in the control code).

Our approach builds on leading edge automatic loop parallelization and transformation techniques based on the *polyhedral model* [3], [4], [5], and it is applicable to a much wider class of programs (namely imperfectly nested loops with affine bounds and index functions) than previously published works [6], [7], [8], [9]. This is the reason why we call this method *polyhedral bubble insertion*.

This article is organized as follows, Section II provides an in depth description of the problem we tackle in this work, and emphasizes the shortcomings of existing approaches. Section III aims at summarizing the principles of program transformations and analysis in the polyhedral framework. Sections IV and V present our pipeline legality analysis and our pipeline schedule correction technique and Section VI provides a quantitative analysis of our results. In section VII we present relevant related work, and highlight the novelty of our contribution. Conclusion and future work are described in section VIII.

II. Motivations

A. Loop pipelining in HLS tools

The goal of this section is to present and motivate the problem we address in this work, that is *nested loop pipelining*. To help the reader understand our contributions, we will use throughout the remaining of this work a running toy loop-nest example shown in Figure 1; it consists in a double nested

978-1-4577-1740-6/11/$26.00

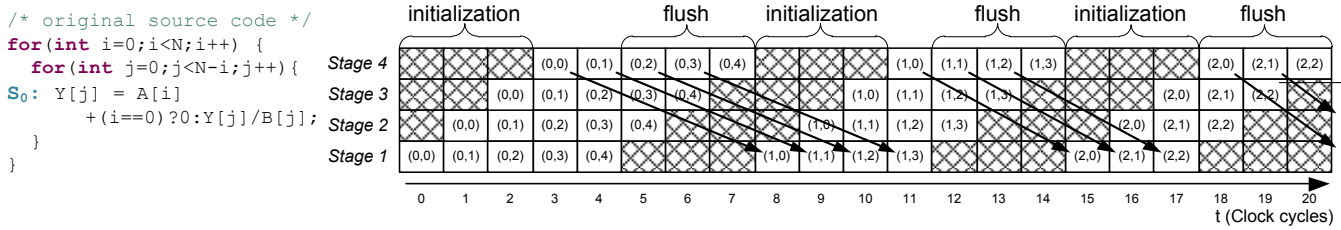


Fig. 2. Motivating example, with a representation of its pipelined execution for $N = 5$, $II = 1$ and $\Delta = 4$. The arrows reprensent dependences between operations.

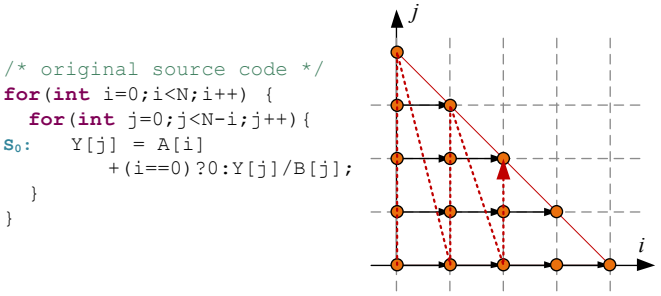


Fig. 1. Motivating example, with its iteration domain and data dependencies (black arrows) for $N = 5$. The red dashed arrow represents the execution order. .

loop operating on a triangular *iteration domain* – the iteration domain of a loop is the set of values taken by its loop indices[1].

The reader can observe that the inner loop (along the j index) exhibits no dependencies between calculations done at different iterations (also called *loop carried dependencies*). As a consequence, one can *pipeline* the execution of this loop, by overlapping the execution of several iterations. However, there exists a dependence between i iterations, since `Y[j]` (left-hand side) depends on `Y[j]` (right-hand side) that was modified during the previous i iteration. Therefore, overlapping the execution of two successive i iterations has to be done with care, in order to respect this dependency.

Loop pipelining is characterized by two important parameters:

- The *initiation interval* (denoted II in the following), that corresponds to the number of clock cycles separating the execution of two loop iterations.
- The *latency* (denoted Δ) that gives the number of clock cycles required to completely execute one iteration of the loop.

As an illustration, Figure 2 depicts the pipelined execution of the example of Figure 1 with an initiation interval $II = 1$ and a latency of $\Delta = 4$. In practice the value of II is constrained by two factors:

- the presence of loop carried dependencies, which prevents loop iterations to be completely overlapped;
- resource constraints on the available hardware since for a complete pipelined execution, each operation executed in the loop has to be mapped on its own hardware functional unit.

Because it helps maximizing the computation throughput and because it improves hardware utilization, loop pipelining is a key transformation in High-Level Synthesis tools. Besides, as designers generally seek to get the best performance from their implementation, fully pipelining the loop (that is initiating a new inner loop iteration every cycle by choosing $II = 1$) is a very common practice. The use of very deep pipeline is even more common when targeting FPGAs devices, as it is often a way to compensate for their relative lower clock speed compared to ASICs. Besides, because the register-cost overhead of pipelining can be easily absorbed by the large amount of flip-flop available on most devices, deeply pipelining FPGA datapath is a very profitable optimization.

However, the performance improvements obtained through pipelining are often hindered by the fact that these tools rely on very basic data-dependency analysis algorithms, and hence they may fail to detect when such a pipelined execution is possible, especially when the inner loop involves complex memory access patterns.

To help designers cope with these limitations, most tools hence offer the ability to bypass part of this conservative dependency analysis through the use of compiler directives (generally in the form of `#pragma`). These directives force the tool to ignore user-specified memory references in its dependency analysis. Of course, this possibility comes at the risk of generating an illegal pipelined schedule and then an incorrect circuit, and hence puts the burden to the designer.

B. The Pipeline Latency Overhead

For loops with large iteration count – we call *loop iteration count* the number of iterations executed by a loop –, the impact of the pipeline latency on performance can be neglected, and the hardware is then almost 100% utilized. However, whenever the iteration count of the loop becomes comparable to its latency Δ, one may observe a very significant performance degradation, as the pipeline flushing phases dominate the execution time. This is the case of our example in Figure 2. For a value of $N = 5$ and $\Delta = 4$, we obtain a hardware

[1]This toy loop is actually a simplified excerpt from the QR factorization algorithm.

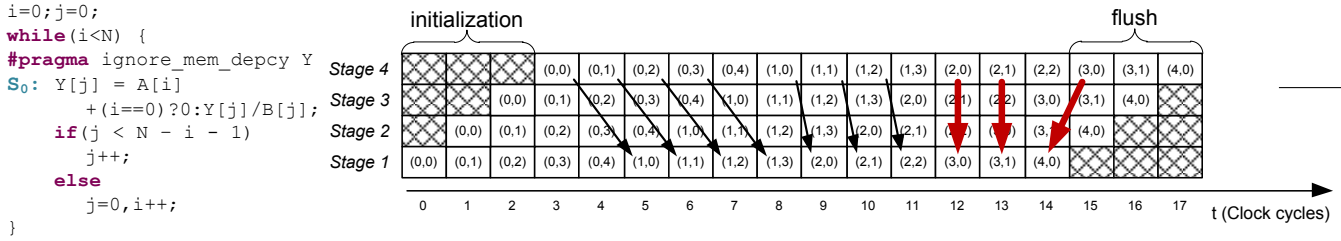


Fig. 3. Illegal nested loop pipelining for $N = 5$, $II = 1$ and $\Delta = 4$. Bold arrows show broken dependences.

utilization rate of only 50%. Indeed, the dependency between two successive i iterations prevents the end of the inner loop pipeline to be overlapped with the beginning of the next pipeline.

Returning to our example, had it to be mapped to custom hardware by experienced designers, it would have certainly reached a hardware utilization close to 100% thanks to a handcrafted schedule, in which the execution of successive iterations of the i loop would have been carefully overlapped.

C. Nested loop pipelining & coalescing

It turns out that such an optimization actually corresponds to a *nested loop pipelining* as initially proposed by Doshi et al. [6]. Such *nested loop pipelining* can be realized through a *loop coalescing* transformation, that flattens a loop nest into a single loop that scans the original loop nest domain, and then pipelines the new loop.

It is worth noticing that *nested loop pipelining* was only studied in the scope of a very restrictive subset of loop nests (perfectly nested loop with constant bounds and uniform dependencies) or with relatively imprecise dependency information, which significantly restricts its applicability and/or efficiency. While these restrictions may seem over precautious, it happens that implementing nested loop pipelining (and more particularly enforcing its correctness) is far from trivial and requires a lot of attention.

As an example, Figure 3 shows a coalesced version of the loop nest of Figure 1. Here, because the array accesses in the coalesced version are now more difficult to analyze (they do not depend on loop indices as in Figure 1), we are tempted to bypass some of the dependence analysis through a `#pragma ignore_mem_dpcy Y` directive to enable loop pipelining, as explained in subsection II-A. This directive tells the scheduler to ignore data dependences related to the `Y[j]` array accesses in the statement following the directive. Without such directive, the conservative dependence analysis forbids pipelining.

While this scheduling seems correct at the first glance, it appears that some Read after Write dependencies are violated when $i \geq 3$, as shown in Figure 3. For example the memory read operation on `Y[0]` of $(i = 3, j = 0)$ scheduled at $t = 12$ happens before `Y[0]` is updated by the write operation of $(i = 2, j = 0)$ also scheduled at $t = 12$ on the last stage.

As an illustration of this difficulty, among the numerous commercial and academic C to hardware tools that we have evalutated, only one of them (let us call it *Trebuchet-C++*) actually provides the ability to perform such automatic nested loop pipelining. However, its implementation in the tool suffers from severe flaws and generates illegal schedules whenever the domain is not rectangular and/or has non constant loop bounds. From what we understand, even without directives to ignore data dependences, the tool fails for the very same reasons as depicted in Figure 3, that is the tool's analysis is assuming that there are no dependencies carried by the outerloop over the `Y` array.

It can be argued that a simple solution for handling non rectangular loop domains is to resort to a *linearization* of the loop nest prior to pipelining. This linearization consists in padding the iteration domain with *wait states* iterations so as to ensure that the domain scanned by the loop nest is rectangular. This approach turns out to be very inefficient in practice: in our example, the execution overhead (50 %) would be as large as for the non nested pipeline case, beside it does only solve the problem in the case of uniform data dependencies.

D. Contributions of this work

In what follows, we provide a formalization of the conditions under which nested loop pipelining is legal w.r.t data dependencies in the case of imperfectly nested loops with affine dependencies (so called SCoPs [3]), where exact (i.e. iteration wise) data dependence information is available.

In addition to this legality check, we also propose a technique to *correct* an a priori illegal nested pipeline schedule by inserting *wait states* in the coalesced loop, so as to derive the most efficient *legal* pipelined schedule. These wait states correspond to properly inserted *bubbles* in the pipeline, so the name *polyhedral bubble insertion* of our method.

Finally, to enable experimentation and to remain as vendor independent as possible, we propose an implementation of the polyhedral bubble insertion in the context of a source-to-source compiler that can be used as a preprocessing tool to be used in front of third parties HLS compilers.

III. Background

In order to do the analysis and the cycle-accurate schedule correction, an iteration-wise dependence analysis as well as a new intermediate representation of loops is necessary. The

polyhedral model is a robust mathematical framework to represent loops; it also comes with a set of techniques to analyze and transform loops and to generate code. In this section, we briefly present the background needed to understand our method.

A. Structure and Limitations

The polyhedral model is a representation of a subset of programs called Static Control Parts (SCoPs), or alternatively Affine Control Loops (ACLs). Such programs are composed only of loop and conditional control structures and the only allowed statements are array assignments of arbitrary expressions with array reads (scalar variables are special cases viewed as zero-dimensional arrays). The loop bounds, the conditions and array subscripts have to be affine expressions of loop indexes and parameters.

Each statement S surrounded by n loops in a SCoP has an associated domain $\mathcal{D}_S \subseteq \mathbb{Z}^n$. The domain $\mathcal{D}_S$ represents the set of values the indices of the loops surrounding S can take. Each vector of values in $\mathcal{D}_S$ is called an *iteration vector*, and $\mathcal{D}_S$ is called the *iteration domain* of S. $\mathcal{D}_S$ is defined by a set of affine constraints, i.e. the set of loop bounds and conditionals on these indexes. In what follows, we call *operation* a particular statement iteration, i.e., a statement with a given iteration vector. Figure 1 shows the graphical representation of such a domain, where each full circle represents an operation. The domain's constraints for the only statement of Figure 1 are as follows:

$$\mathcal{D} = \{i, j | 0 \leq i < N \wedge 0 \leq j < N - i\} \quad .$$

The polyhedral model is limited to the aforementioned class of programs. This class can be however extended to a larger class of programs at the price of a loss of accuracy in the dependance analysis [10], [11].

B. Dependences and Scheduling

The real strength of the polyhedral model is its capacity to handle iteration wise dependence analysis on arrays [12]. The goal of dependence analysis is to answer questions of the type *"what is the statement that produced the value being read at current operation, and for what iteration vector?"* For example, in the program of Figure 1, what is the operation that wrote the last value of the right-hand side reference `Y[j]`?

Iterations of a statement in a loop nest can be ordered by the lexicographic order of their iteration vectors. The combination of the lexicographic order and the textual order gives the precedence order (noted $\succ$) of operations, that gives the execution order of operations in a loop nest. When considering sequential loop nests, the precedence order is total.

The precedence order allows an exact answer to be given to the previous question: *"the operation that last modified an array reference in an operation is just the latest one in the precedence order."* In the example of Figure 1, the operation that modified right-hand side reference `Y[j]` in operation $S_0(i, j)$ is just the same statement of the loop, when it was executed at previous iteration $S_0(i-1, j)$.

In the polyhedral model, building this precedence order can be done exactly. Therefore, transformations of the loop execution order, also known as *scheduling* transformations, can be constrained to enforce dataflow dependences. This feature may be used to check the legality of a given transformation, but also to automatically compute the space of all possible transformations, in order to find the "best" one. However this is not the topic of this paper, and the reader is referred to Feautrier [13] and Pouchet et al. [5] for more details.

C. Code generation

Once a loop nest has been scheduled (for example, to incorporate some pipelining), the last step of source-to-source transformation consists in re-generating a sequential code. Two approaches to solve this problem dominate in the litterature. The first one was developed by Quillere and al. [14] and later extended by Bastoul in the context of the ClooG software [3]. ClooG allows regenerated loops to be guardless, thus avoiding useless iterations at the price of an increase in code size. With the same goal, the code generator in the Omega project also tries to regenerate guardless loops, but also provides options to find a trade-off between code size and guards [15].

The second approach, developed by Boulet et al. [16] aims at generating code without loops. The principle is to determine during one iteration the value of the next iteration vector, until all the iteration domain has been visited. Since this second approach behaves like a finite state machine, it si believed to be it is more suited for hardware implementation [17], though there is still very few quantitative evidences to back-up this claim.

IV. Legality Check

In this section, we propose sufficient conditions for ensuring that a given loop coalescing transformation is legal w.r.t to the data-dependencies of the program.

Consider a sink reference to an array (i.e. a right-hand side array reference in a statement), and let $\vec{y}$ denote its iteration vector. Let $\vec{x}$ be the iteration vector of the source reference for this array reference. Let us write d the function that maps $\vec{y}$ to $\vec{x}$, so that $\vec{x} = d(\vec{y})$.

We define Δ as the highest latency, in the pipeline datapath, between a read and a write inducing a dependence. We can formulate the conditions under which a loop coalescing is legal w.r.t to this data-dependency as follows: for a pipelined schedule with a latency of Δ, the coalescing will violate data dependencies when the distance (in number of iteration points) between the production of the value (at iteration $\vec{x}$, the source) and its use (at iteration $\vec{y}$, the sink) is less than Δ.

This condition is trivially enforced in one particular case, that is when the loops to be coalesced do not carry any dependences, that is when the loops are parallel. This is possible since one may want to pipeline only the $n-1$ inner loops of the loop nest in which the dependences are only carried by the outermost loop. In such a case, the pipeline is flushed at each step of the outermost loop, hence the latency does not break any dependence.

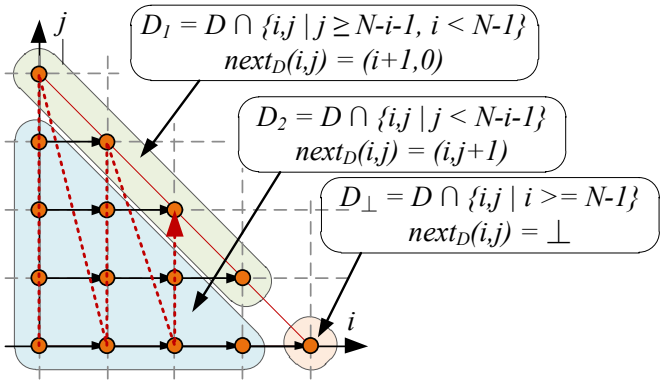


Fig. 4. Sub-domains D_1 and D_2 have different expressions for their immediate successor.

Let p be the depth of the loop that carries a dependence, the coalescing is ensured to be legal if the loop to be coalesced are at a depth greater than p. In practice, the innermost loop is the only depth carrying no dependence, as shown in the example of Figure 1.

Determining if a coalescing is legal then requires a more precise analysis, by computing the number of points between a source iteration $\vec{x}$ and its sink $\vec{y}$. This indeed amounts to count the number of integral points inside a parametric polyhedral domain and corresponds to the *rank* function as proposed by Turjan et al. [18], for which we can obtain a closed form expression using the Barvinok library [19]. However these expressions are in the form of parametric multivariate pseudo-polynomials, and checking whether the value of such polynomials admits a given lower bound is impossible in the general case.

Because of this limitation, we propose another technique which does not involve any polyhedral counting operation. In this approach, we construct a function $next^{\Delta}_{\mathcal{D}}(\vec{x})$ that computes for a given iteration vector $\vec{x}$ its successor Δ iterations away in the coalesced loop nest's iteration domain $\mathcal{D}$. We then check that all the sink iteration vectors $\vec{y} = d^{-1}(\vec{x})$ of the dependency d are such that $\vec{y} \succeq next^{\Delta}_{\mathcal{D}}(\vec{x})$. In other words, we make sure that the value produced at iteration $\vec{x}$ is used at least Δ iterations later.

The only difficulty in this legality check lies in the construction of the $next^{\Delta}_{\mathcal{D}}(\vec{x})$ function. This is the problem we address in the following subsection.

A. Constructing the $next^{\Delta}_{\mathcal{D}}(\vec{x})$ function

We will derive the $next^{\Delta}_{\mathcal{D}}$ function by leveraging on a method presented by Boulet et al[16] to compute the immediate successor in $\mathcal{D}$ of an iteration vector $\vec{x}$ according to the lexicographical order. This function is expressed as a solution of a lexicographic minimization problem on a parametric domain made of all successors of $\vec{x}$.

The algorithm works as follows: we start by building the set of points for which the immediate successor belongs to the same innermost loop (say at depth p). This set is represented as D_2 in the example of Figure 4. We then do the same for the set of points of $\mathcal{D}$ for which no successors were found at previous step, but this time we look for their immediate successors along the loop at depth $p-1$ as shown by the domain D_1 in Figure 4. This procedure is then repeated until all dimensions of the domain have been covered by the analysis. At the end, the remaining points are the lexicographic maximum (that is the end) of the domain, and their successor is noted as $\perp$ ($D_{\perp}$ on Figure 4).

The domains involved in this algorithm are parameterized, therefore the approach requires the use of a Parametric Integer Linear Programming solver [20], [16] to obtain a solution which is in the form of a quasi affine mapping function that defines the sequencing relation. Because it is a quasi-affine function[2], and because we only need to look for a constant number of iterations ahead (the latency of the pipeline that we call Δ), we can easily build the $next^{\Delta}_{\mathcal{D}}$ function. This is done by applying the function to itself Δ times as shown in Equ. (1) :

$$next^{\Delta}_{\mathcal{D}}(\vec{x}) = \overbrace{next_{\mathcal{D}} \bullet next_{\mathcal{D}} \bullet \ldots \bullet next_{\mathcal{D}}}^{\Delta}(\vec{x}) \quad . \qquad (1)$$

Example: Let us compute the $next_{\mathcal{D}}(\vec{x})$ predicate for the example of Figure 4, where we have $\vec{x} = (i,j)$

$$next_{\mathcal{D}}(i,j) = \begin{cases} (i,j+1) & \text{if} & j < N-i-1 \\ (i+1,0) & \text{elseif} & i < N-1 \\ \perp & \text{otherwise} & \end{cases}$$

Note that $\perp$ represents the absence of successor in the loop. Applying the relation four times to itself we then obtain the $next^4_{\mathcal{D}}(i,j)$ predicate, which is given by the mapping below :

$$next^4_{\mathcal{D}}(i,j) =$$

$$\begin{cases} (i,j+4) & \text{if} & j \leq N-i-5 \\ (i+1,3) & \text{elseif} & i \leq N-5 \wedge j = N-i-1 \\ (i+1,2) & \text{elseif} & i \leq N-4 \wedge j = N-i-2 \\ (i+1,1) & \text{elseif} & i \leq N-3 \wedge j = N-i-3 \\ (i+1,0) & \text{elseif} & i \leq N-4 \wedge j = N-i-4 \\ (N-1,0) & \text{elseif} & i = N-3 \wedge j = 1 \wedge N \geq 3 \\ (N-2,0) & \text{elseif} & i = N-4 \wedge j = 3 \wedge N \geq 4 \\ \perp & \text{else} & \end{cases} \qquad (2)$$

B. Building the violated dependency set

As mentioned previously, a given dependency is enforced by the coalesced loop iff we have $\vec{y} \succeq next^{\Delta}_{\mathcal{D}}(\vec{x})$ with $\vec{y} = d^{-1}(\vec{x})$. When $next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\}$, that is when the successor $\Delta - 1$ iterations later is out of the iteration domain, the dependence is obviously broken. We can then build $\mathcal{D}^{\dagger}$ the domain containing all the iterations sourcing one of these violated dependencies, using the equation below

$$\mathcal{D}^{\dagger} = \left\{ \vec{x} \in \mathcal{D}_{src} \;\middle|\; \begin{array}{l} d^{-1}(\vec{x}) \prec next^{\Delta}_{\mathcal{D}}(\vec{x}) \\ \text{or } next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\} \end{array} \right\} \qquad (3)$$

[2]Quasi affine function are affine functions where division (or modulo) by an integer constant are allowed.

where $\mathcal{D}_{src}$ is the set of sources of a dependency in $\mathcal{D}$.

It is important to note that in case of a parameterized domain, the set of these iterations may itself be a parameterized domain. Checking the legality of a nested loop pipelining then sums up to check the emptiness of this parameterized domain, which can easily be done with ISL [21] or Polylib [22].

a) Example: In what follows, we make no difference between relations and functions, following the practice used in the ISL tool. In our example, we have the following dependency relation :

$$d(i,j \rightarrow i',j' : i,j \in \mathcal{D} \wedge i \geq 1 \wedge i' = i-1 \wedge j' = j)$$

which can easily be reverted as

$$d^{-1}(i,j \rightarrow i',j' : i',j' \in \mathcal{D} \wedge i \geq 0 \wedge i' = i+1 \wedge j' = j)$$

which corresponds to the data-dependency.

Using the $next^4_{\mathcal{D}}(i,j)$ function obtained in (2), we can then build the domain $\mathcal{D}^\dagger$ of the source iterations violating a data dependency using (3).

In our example, and after resorting to the simplification of this polyhedral domain thanks to a polyhedral library [21], we then obtain :

$$\mathcal{D}^\dagger = \{i,j | (i,j) \in \mathcal{D} \wedge N-4 < i < N-1 \wedge j < N-i-1\}$$

When we substitute N by 5 (the chosen value in our example), we have $\mathcal{D}^\dagger = \{(2,0),(2,1),(3,0)\}$, which is the set of points that causes a dependency violation in Figure 3.

V. Bubble Insertion

While a legality condition is an important step toward automated nested loop pipelining, it is possible to do better by *correcting* a given schedule to make the coalescing legal. Our idea is to determine *at compile time* an iteration domain where *wait states*, or *bubbles*, are inserted in order to stall the pipeline to make sure that the coalesced loop execution is legal w.r.t data dependencies. Of course we want this set to have the smallest possible impact on performance, both in terms of number of cycles, and in terms of overhead caused by extra guards and housekeeping code.

We already know from the previous subsection the domain $\mathcal{D}^\dagger$ of all iterations whose source violates the dependency relation. To correct the pipeline schedule we can insert additional wait-state iterations in the domain scanned by the coalesced loop. These wait state iterations should be inserted between the source and the sink iterations of the violated dependency. One obvious solution is to add these extra iterations at the end of the inner loop enclosing the source iteration, so that this extra cycle may benefit to all potential source iteration within this innermost loop.

The key question in this problem is to determine how many of such wait states are actually required to fix the schedule, as adding a single wait state in a loop may incidentally fix/correct several violated data-dependency. In the following we propose a simple technique to solve the problem.

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:    Y[j] = A[i]
         +(i==0)?0:Y[j]/B[j];
   if((i>N-4&&j<N-i+3&&i<N-1)
      ||j<N-i-1)
       j++;
   else
       j=0,i++;
}
```

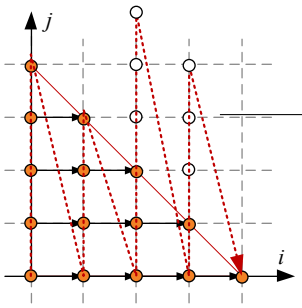


Fig. 5. Corrected pipeline (conservative) for $N = 5$ and $\Delta = 4$. White points correspond to pipeline epilogue inserted only for the iterations that need it.

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:    Y[j] = A[i]
         +(i==0)?0:Y[j]/B[j];
   if((i>N-4&&j<3&&i<N-1)
      ||j<N-i-1)
       j++;
   else
       j=0,i++;
}
```

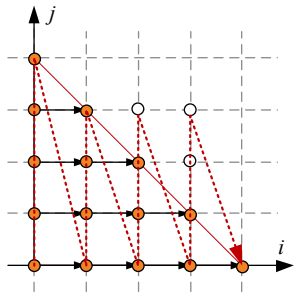


Fig. 6. Corrected pipeline (optimized) for $N = 5$ and $\Delta = 4$. White points correspond to the inserted pipeline bubbles in the iteration domain.

The simplest solution is to pad every inner loop containing an iteration in $\mathcal{D}^\dagger$ with $\Delta - 1$ wait-states. As a matter of fact, this amounts to recreate the whole epilogue of the pipelined loop, but only for the outer loops that actually need it. The approach is illustrated in Figure 5, but turns out to be too conservative. For example, the reader will notice that the inner loops for indices $i = 2$ in the example of Figure 1 do not actually need $\Delta - 1 = 3$ additional cycles. In that case only one cycle of wait state is needed, and similarly, for $i = 3$, only two cycles are needed.

This solution is obviously not optimal, as one could easily find a better correction (that is with fewer bubbles), as the one shown in Figure 6. In this particular example, the lower overhead (in terms of wait-state) does not come at the price of an increase in the complexity of the code. This is however not the case in general, and there exist subtle trade-offs between the reduction of bubble count and the complexity of the control logic. We are currently investigating this topic.

VI. Results and Validation

In this section, we describe how the transformation is being implemented within a compiler framework, and provide quantitative data showing that the approach is practical and lead to significant performance improvements at the price of a moderate increase in area.

A. The Gecos source to source compiler

GeCoS (Generic Compiler Suite) is an open source-to-source compiler infrastructure [23] integrated within the Eclipse framework and entirely based on Model Driven Software Development tools. GeCoS is specifically targeted to HLS back-ends, and provides (among other features) built-in support for Mentor Algorithmic Data types C++ templates (`ac_int<>`, etc.).

GeCoS also provides a loop transformation framework based on the polyhedral model, that extensively uses third party libraries (ISL for manipulating polyhedral domains [24] and solving parametric integer linear problems, and Cloog [3] for polyhedral code generation). All the transformations presented in this work have been implemented within this framework.

B. Implementing the loop coalescing transformation

Implementing the loop coalescing transformation amounts in a rewriting of the loop nest structure into a software finite state machine expressed in a while loop. There are two possible approach for implementing this rewriting.

The first approach uses the Control Flow Graph (CFG) corresponding to the loop nest as an input. The problem in this approach is that the automaton is built from an implicit representation of the iteration domain rather than from its formal representation as a polyhedron. As a consequence the resulting automaton contains extra idle states which do not correspond to an actual iteration of the loop nest. On the other hand the main advantage of this approach is its simplicity. From what we understand, this is the approach followed by our reference HLS tool when implementing the *nested loop pipelining* transformation.

The second approach follows the approach of Boulet et al. [16] and consists in building a finite state machine directly out of the loop nest iteration domain[3] using the $next_{\mathcal{D}}(\vec{x})$ mapping introduced in Section IV. Because it is the only way to ensure that the generated code visits the *exact* loop nest iteration domain, it is actually more efficient than the approach based on the CFG. However, the resulting code tends to be more complex and induces area overhead as the number of guards grows as the number of dimension of domain increases and its shape gets more complex.

Because one does generally not want to coalesce the full loop nest, our approach allows the designer to choose (by using a source code directive) how many of the inner most loops should be coalesced, and then use a combination of Cloog with the algorithm of Boulet & Feautrier to regenerate the transformed loop nest structure.

C. Experimental results

The efficiency of our approach is obviously very sensitive to the trip count in inner loops sizes. As such it is very efficient for improving the performance of non perfectly tiled codes and of non rectangular domains. It is therefore more sensitive to the iteration domains size and domain shapes than to the structure of the application itself.

In this work, we limited ourself to two representative kernels (QR-Cordic, MatMul described below) that provide a good illustration the benefits and drawbacks of the approach.

- The *QR decomposition* is a key building block for wireless MIMO communication systems. The QR kernel operates over (small) non rectangular 3-dimensional iteration domain and is a good candidate for the approach. In the QR, only the most inner loop (index k) is parallel, and this loop has a non constant iteration count (Indeed the two innermost loops of the QR kernel resembles a lot our running toy example). As a consequence, direct nested pipelining causes dependence violation, and the kernel must therefore undergo a *bubble insertion step*.
- A *Matrix Multiplication* kernel, in which we performed a loop interchange to enable the pipelining of the 2 innermost loops. In this case the iteration domain is very simple (rectangular), but we allow in some cases the matrix sizes to be parameterized (i.e not known at compile time).

Because our reference HLS tool does not support division nor square root operations, we replaced these operations in the QR algorithm with deeply pipelined multipliers. We insist on the fact that this modification does not impact the relevance of the results given below, since our coalescing transformation only impacts the loop control, the body being left untouched.

For each of these kernels, we used varying fixed and parameterized iteration counts, and also used different fixed point arithmetic wordlength sizes for the loop body operations so as to be able to precisely quantify the trade-off between performance improvement (thanks to nested pipeline) and area overhead (because of extra control cost).

For each application instance, we compared the results obtained when using :

- Simple loop pipelining by our reference HLS tool.
- Nested loop pipelining by our reference HLS tool.
- Nested loop pipelining through loop coalescing and bubble insertion.

For all examples, we derived deeply pipelined datapaths (with $II = 1$ in all cases) and with latency values varying from 4 to 6 in the case of Matrix Multiplication, and from 9 to 12 in the case of the QR factorization depending on the fixed point encoding.

We provide three metrics of comparison: the total accelerator area cost (in LUT and registers), the number of clock cycles required to execute the program, and the clock frequency obtained by the design after place and route. All the results were obtained for an Altera Stratix-IV device with fastest speed-grade, and are given in Table I.

The quantitative evaluation of the area overhead induced by the use of nested pipelining is provided in Figure 7. Our results show that this overhead remains limited and even negligible when large functional units are being used (in the figure, `<a,b>` stands for a `a` bit wide fixed point format with

[3]It is to note that the method can accommodate with imperfectly nested loops thanks to the use of additional (scalar) dimensions which model textual ordering as in the Cloog library [3].

Benchmark	Latency	Size	LUTs		Registers		DSPs		Freq (MHz)		Clock Cycles	
			HLS	Coal.	HLS	Coal.	HLS	Coal.	HLS	Coal.	HLS	Coal.
MM<48,16>	6 cycles	param	512	579	657	677	10	10	160	164	n.a.	n.a.
		128	437	392	542	458	10	10	161	163	2114304	2097157
		32	388	351	486	426	10	10	164	160	33984	32773
		8	333	313	429	442	10	10	164	164	624	517
MM<24,6>	4 cycles	32	239	200	170	130	4	4	240	250	33920	32771
QR<48,16>	12 cycles	param	999	1190	1114	2169	10	10	166	166	n.a.	n.a.
		128	1944	3262	7209	7891	10	10	162	164	902208	797050
		32	1229	1534	2562	3230	10	10	167	165	23312	17370
		8	1018	1951	1662	2297	10	10	166	167	868	746
QR<24,6>	9 cycles	32	620	823	1064	1240	4	4	231	229	20336	15552

TABLE I
PERFORMANCE AND AREA COST FOR OUR NESTED PIPELINE IMPLEMENTATIONS

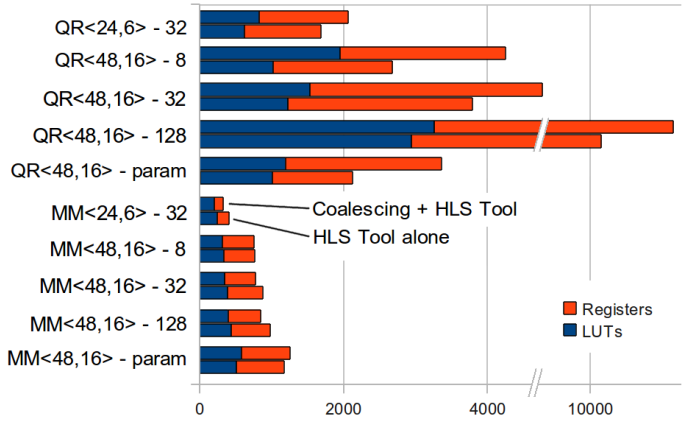


Fig. 7. Area overhead due to the loop coalescing and bubble insertion, this overhead is caused by extra guards on loop indices.

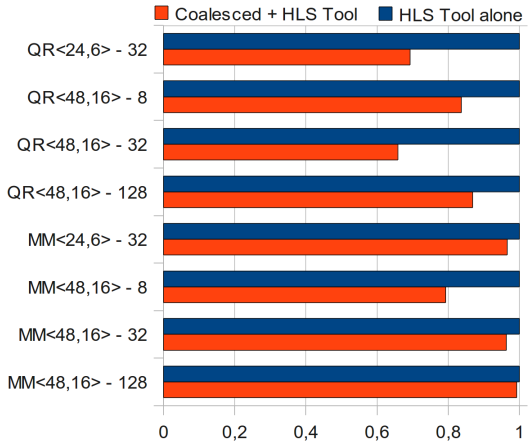


Fig. 8. Normalized execution time (in clock cycles) of the two innermost coalesced pipelined loops (with bubbles for QR) w.r.t to the non-coalesced pipelined loop.

b bit devoted to the integer part). Also, the approach does not significantly impact the clock frequency (less than 5% difference in all cases).

The improvement in execution time due to latency hiding are given in Figure 8. Here one can observe that the efficiency of the approach is highly dependant on the loops iteration count. While the execution time can decrease by up to 34% in some case, the benefits quickly decrease as the domain sizes grow. For larger iteration counts, the performance improvement hardly compensates the area overhead.

One interesting observation is that when no correction is needed (e.g constant size matrix multiplication) our coalescing transformation is more efficient in term of both performance and area than the nested pipeline feature provided with the tool, a result which is easy to explain (see VI-B).

In addition to this quantitative analysis, it is also interesting to point out which examples did cause our reference leading edge commercial HLS tools to either find a good nested loop pipelined schedule or to generate an illegal schedule violating the semantic of the initial program. For the *QR* example, the reference tool would systematically fail to generate a legal nested pipeline schedule for the algorithm. Furthermore it gives an illegal schedule whenever its iteration domain is parameterized.

Benchmark	$next$	$next^{15}$
ADI Core	1391	71517
Block Based FIR	697	131284
Burg2	1166	36757
Forward Substitution	59	734
Hybrid Jacobi Gauss Seidel	15	3065
Matrix Product	187	29245
QR Given Decomposition	72	4554
SOR 2D	90	30151

TABLE II
$next$ AND $next^{15}$ RUNTIME EVALUATION (IN *ms*)

Last, we also evaluated the runtime needed to perform the $next^k$ operations for several examples. The goal is to demonstrate that the approach is practical in the context of a HLS tool. Results are given in Table II, and show that the runtime (in *ms*) remains acceptable in most cases.

VII. Related work and discussion

A. Loop pipelining in hardware synthesis

Earlier work on systolic architectures addressed the problem of fine grain parallelism extraction. Among others Derrien et al. [8] proposed to use iteration domain partitioning to help combine operation level (pipeline) and loop level parallelism. A somewhat similar problem was addressed by Teich and al.[9] who proposed to combine modulo scheduling with loop-level parallelization techniques. The main limitation of these contributions is that they only support *one-dimensional schedules* [25], which significantly limit their applicability.

A recent work by Alias et al. [26] tackles a problem very similar to ours, as they try to address the problem of generating efficient nested loop pipelined hardware accelerators leveraging custom floating point datapaths. Their approach (also based on the polyhedral model) consists in finding a parallel hyperplane for the loop nest, and then derive a tiling (hyperplanes and tile sizes) which is chosen such that a pipeline of depth Δ is legal. The approach only targets perfectly nested loop and also requires incomplete tiles to be padded to behave like full-tiles. Besides they restrict themselves to uniform dependencies, so as to guarantee that the reuse distance (i.e the number of points separating a source and a sink) is always constant for a given tile size. In contrast, our framework is more general and supports imperfectly nested loops with non-uniform (i.e affine) dependencies. In addition, in the case of tiled iteration domains, we can provide a more precise correction (in terms of extra bubbles) that does not requires padding all incomplete tiles.

The Compaan/Laura [18] toolset takes another view on the problem, as it does not try to find a global schedule for the program statements. Instead, each statement of the program is mapped on its own process. Dependencies between statements are then materialized as communication buffers which follow the so-called Polyhedral Process Network semantic [27]. Because the causality of the schedule is enforced by the availability of data on the channel output, there is no need for taking statement execution latency into account in the process schedule [28]. On the other hand, the approach suffers significant area cost overhead as each statement requires its own hardware controller plus possibly complex reordering memory structure. To our opinion, the approach is geared toward task level parallelism rather than toward fine grain parallelism/pipeline.

B. Nested loop software pipelining

Software pipelining has proved to be a key optimization for leveraging the instruction level parallelism available in most compute intensive kernels. Since its introduction by Lam et al. [29] a lot of work has been carried out on the topic (a survey is out of scope of this work). Two directions have mainly been addressed:

- Many contributions have tried to extends software pipelining applicability to wider classes of program structures, by taking control flow into consideration [30].
- The main other research direction has focused on integrating new architectural specificities and/or additional constraints when trying to solve the optimal software pipelining problem [31].

Among these numerous contributions, some of them have been tackling problems very close to ours.

First, Rong et al. [7] have already studied the problem of nested loop software pipelining. Their goal is clearly the same as ours, except that they do restrict themselves to a narrow subset of loop (only constant bound rectangular domains) and do not leverage exact instance-wise dependence information. Besides they do not address the problem from a hardware synthesis point of view. In this work, we tackle the problem for a wider class of programs (known as Static Control Parts), and also we relate the problem to *loop coalescing*.

Another related contribution is the work of Fellahi et al [32], who address the problem of prologue/epilogue merging in sequences of software pipelined loops. Their work is also motivated by the fact that the software pipeline overhead tends to be a severe limitation as many embedded-multimedia algorithms exhibit *low trip count* loops. Again, our approach differs from theirs in the scope of its applicability, as we are able to deal with loop nests (not only sequences of loops), and as we solve the problem in the context of HLS tools at the source level through a loop coalescing transformation. On the contrary their approach handles the problem at machine code level, which is not possible in our context.

C. Loop coalescing and loop collapsing

Loop coalescing was initially used in the context of parallelizing compilers, for reducing of synchronization overhead [33]. Since synchronization occurs at the end of each innermost loop, coalescing loops reduces the number of synchronization during the program execution. Such an approach is quite similar to ours (indeed, one could see the flushing of the innermost loop pipeline as a kind of synchronization operation). However, in our case we can benefit from an exact timing model of the synchronization overhead, which we can be used to remove unnecessary synchronization steps.

D. Correcting illegal loop transformations

The idea of applying a correction on a schedule as a post-transformation step is not new, and was introduced by Bastoul et al [34]. Their idea was to first look for interesting combination of loop transformations (be they legal or not), and then try to fix possible illegal schedule instances through the use of loop shifting transformations. Their result was later extended by Vasilache et al. [35], who considered a wider space of correcting transformations.

Our work differs from theirs in that we do not propose to modify the existing schedule, but rather add artifact statements whose goal is to model so called *wait state* operations, which will then make loop coalescing legal w.r.t data dependencies.

VIII. Conclusion

In this paper, we have proposed a new technique for supporting nested loop software pipelining in C to hardware synthesis

tools. The approach extends previous work by considering a more general class of loops nests. In particular we propose a nested pipeline legality check that can be combined with a compile time bubble insertion mechanism to enforce causality in the pipelined schedule. Our nested loop pipelining technique was implemented as a proof of concept and our preliminary experimental results show promising results for nested loops operating on small iteration domains (up to 30 % execution time reduction in terms of clock cycles, with a limited area overhead).

As a side-note, we believe that our approach can easily be adapted to be used in a more classical optimizing compiler back-ends. Of course, our approach would only makes sense for deeply pipelined VLIW machines with many functional units. In that case we simply need to use the value of the loop body initiation interval as an additional information to determine which dependencies may be violated.

ACKNOWLEDGEMENT

The authors would like to thanks Sven Verdoolaege, Cedric Bastoul and all the contributors to the wonderful pieces of software that are ISL and ClooG. This work was founded by the INRIA-STMicroelectronic Nano2012 project.

REFERENCES

[1] M. Graphics, "Catapult-c synthesis," http://www.mentor.com.

[2] "Autoesl design technologies," http://www.autoesl.com/.

[3] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, Sep. 2004, pp. 7–16.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "PLuTo: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Tucson, AZ: ACM, June 2008.

[5] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. Tucson, Arizona: ACM Press, June 2008, pp. 90–100.

[6] K. Muthukumar and G. Doshi, "Software pipelining of nested loops," in *Proceedings of the 10th International Conference on Compiler Construction*, ser. CC '01. London, UK: Springer-Verlag, 2001, pp. 165–181. [Online]. Available: http://portal.acm.org/citation.cfm?id=647477.727775

[7] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-dimension software pipelining for multidimensional loops," *ACM Trans. Archit. Code Optim.*, vol. 4, March 2007. [Online]. Available: http://doi.acm.org/10.1145/1216544.1216550

[8] S. Derrien, S. Rajopadhye, and S. Kolay, "Combined instruction and loop parallelism in array synthesis for fpgas," in *The 14th International Symposium on System Synthesis. Proceedings.*, 2001, pp. 165 – 170.

[9] J. Teich, L. Thiele, and L. Z. Zhang, "Partitioning processor arrays under resource constraints," *VLSI Signal Processing*, vol. 17, no. 1, pp. 5–20, 1997.

[10] M. W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Compiler Construction*. Springer, 2010, pp. 283–303.

[11] J. F. Collard, D. Barthou, and P. Feautrier, "Fuzzy array dataflow analysis," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 1995, pp. 92–101.

[12] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[13] ——, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[14] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *International Journal of Parallel Programming*, vol. 28, pp. 469–498, 2000. [Online]. Available: http://dx.doi.org/10.1023/A:1007554627716

[15] W. Kelly, W. Pugh, and E. Rosser, "Code generation for multiple mappings," pp. 332–341, February 1995.

[16] P. Boulet and P. Feautrier, "Scanning Polyhedra without Do-loops," in *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 1998, p. 4.

[17] A.-C. Guillou, P. Quinton, and T. Risset, "Hardware Synthesis for Multi-Dimensional Time," in *ASAP*. IEEE Computer Society, 2003, pp. 40–50.

[18] A. Turjan, B. Kienhuis, and E. F. Deprettere, "Classifying interprocess communication in process network representation of nested-loop programs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 2, 2007.

[19] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions," *Algorithmica*, vol. 48, no. 1, pp. 37–66, 2007.

[20] P. Feautrier, "Parametric integer programming," *RAIRO Recherche opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[21] S. Verdoolaege, *Integer Set Library: Manual*, 2010. [Online]. Available: http://www.kotnet.org/ skimo/isl/manual.pdf

[22] D. Wilde, "A library for doing polyhedral operations," IRISA, Tech. Rep., 1993.

[23] The Gecos Source to Source Compiler Infrastructure. [Online]. Available: http://gecos.gforge.inria.fr/

[24] S. Verdoolaege, "ISL: An Integer Set Library for the Polyhedral Model," in *ICMS*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327. Springer, 2010, pp. 299–302.

[25] P. Feautrier, "Some efficient solutions to the affine scheduling problem. I. One-dimensional time," *International journal of parallel programming*, vol. 21, no. 5, pp. 313–347, 1992.

[26] *Automatic Generation of FPGA-Specific Pipelined Accelerators*, Mars 2011.

[27] S. Verdoolaege, *Handbook of Signal Processing Systems*, 1st ed. Heidelberg, Germany: Springer, 2004, ch. Polyhedral process networks.

[28] C. Zissulescu, B. Kienhuis, and E. F. Deprettere, "Increasing Pipelined IP Core Utilization in Process Networks Using Exploration," in *FPL*, ser. Lecture Notes in Computer Science, J. Becker, M. Platzner, and S. Vernalde, Eds., vol. 3203. Springer, 2004, pp. 690–699.

[29] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *PLDI*, 1988, pp. 318–328.

[30] H.-S. Yun, J. Kim, and S.-M. Moon, "Time optimal software pipelining of loops with control flows," *International Journal of Parallel Programming*, vol. 31, pp. 339–391, 2003, 10.1023/A:1027387028481. [Online]. Available: http://dx.doi.org/10.1023/A:1027387028481

[31] C. Akturan and J. M. F., "Caliber: a software pipelining algorithm for clustered embedded vliw processors," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '01. Piscataway, NJ, USA: IEEE Press, 2001, pp. 112–118. [Online]. Available: http://dl.acm.org/citation.cfm?id=603095.603118

[32] M. Fellahi and A. Cohen, "Software Pipelining in Nested Loops with Prolog-Epilog Merging," in *HiPEAC*, ser. Lecture Notes in Computer Science, A. Seznec, J. S. Emer, M. F. P. O'Boyle, M. Martonosi, and T. Ungerer, Eds., vol. 5409. Springer, 2009, pp. 80–94.

[33] M. T. O'Keefe and H. G. Dietz, "Loop Coalescing and Scheduling for Barrier MIMD Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 1060–1064, September 1993. [Online]. Available: http://portal.acm.org/citation.cfm?id=628913.629222

[34] C. Bastoul and P. Feautrier, "Adjusting a program transformation for legality," *Parallel processing letters*, vol. 15, no. 1, pp. 3–17, Mar. 2005, classement CORE : U.

[35] N. Vasilache, A. Cohen, and L.-N. Pouchet, "Automatic correction of loop transformations," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 292–304. [Online]. Available: http://dx.doi.org/10.1109/PACT.2007.17

Appendix C

Personal bibliography

The references given below correspond to all the publictaion that I co-authored and which are referenced in this document. The complete list is available at `http://www.irisa.fr/cosi/HOMEPAGE/Derrien/`.

[1] Naeem Abbas, Steven Derrien, Sanjay Rajopadhye, and Patrice Quinton. Accelerating HMMER on FPGA using Parallel Prefixes and Reductions. In *IEEE International Conference on Field-Programmable Technology (FPT'10)*, pages 37–44, Beijing, China, December 2010. ⟨ pp. 27, 34, 37, 38, 39 ⟩

[2] V. Basupalli, Tomofumi Yuki, Sanjay V. Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton, and David Wonnacott. ompVerify: Polyhedral Analysis for the OpenMP Programmer. In *7th International Workshop on OpenMP, IWOMP 2011*, pages 37–53, 2011. ⟨ p. 74 ⟩

[3] Rayan Chikhi, Steven Derrien, Auguste Noumsi, and Patrice Quinton. Combining Flash Memory and FPGAs to Efficiently Implement a Massively Parallel Algorithm for Content-Based Image Retrieval. In *Reconfigurable Computing: Architectures, Tools and Applications, Third International Workshop, ARC 2007*, pages 247–258, 2007. ⟨ pp. 18, 19, 25 ⟩

[4] Alexandre Cornu, Steven Derrien, and Dominique Lavenier. HLS Tools for FPGA: Faster Development with Better Performance. In *Reconfigurable Computing: Architectures, Tools and Applications - 7th International Symposium, ARC 2011*, pages 67–78, 2011. ⟨ p. 39 ⟩

[5] Vivek D, Tovinakere, Olivier Sentieys, and Steven Derrien. A Polynomial Based Approach to Wakeup Time and Energy Estimation in Power-Gated Logic Clusters. *Journal of Low Power Electronics*, 2011. Accepted for publication on July 30th, 2011. ⟨ p. 51 ⟩

[6] Vivek T D, Olivier Sentieys, and Steven Derrien. Wakeup Time and Wakeup Energy Estimation in Power-Gated Logic Clusters. *International Conference on VLSI Design*, 0:340–345, 2011. ⟨ p. 51 ⟩

[7] Alain Darte, Steven Derrien, and Tanguy Risset. Hardware/Software Interface for Multi-Dimensional Processor Arrays. In *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2005), 23-25 July 2005*, pages 28–35, 2005. ⟨ p. 60 ⟩

[8] Steven Derrien, Anne-Claire Guillou, Patrice Quinton, Tanguy Risset, and Charles Wagner. Automatic Synthesis of Efficient Interfaces for Compiled Regular Architectures. In *Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2002. ⟨ pp. 56, 57 ⟩

[9] Steven Derrien and P. Quinton. Parallezing HMMER for Hardware Acceleration on FPGAs. In *18th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2007)*, pages 10–17, Montreal, Quebec, July 2007. ⟨ pp. 27, 37, 38, 39 ⟩

[10] Steven Derrien and P. Quinton. Hardware Acceleration of HMMER on FPGAs. *Journal of Signal Processing Systems*, 58(1):53–67, October 2010. ⟨ pp. 27, 38, 39 ⟩

[11] Steven Derrien and Sanjay Rajopadhye. FCCMs and the Memory Wall. In *IEEE Symposium on FPGA Custom Computing Machine*, pages 329–330, April 2000. ⟨ p. 56 ⟩

[12] Steven Derrien and Sanjay Rajopadhye. Loop Tiling for Reconfigurable Accelerators. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pages 398–408, 2001. ⟨ pp. 56, 57, 58 ⟩

[13] Steven Derrien and Sanjay Rajopadhye. Energy/Power Estimation of Regular Processor Arrays. In *International Symposium on System Synthesis (ISSS), Kyoto*, pages 50–55, 2002. ⟨ p. 56 ⟩

[14] Steven Derrien, Sanjay Rajopadhye, and Susmita Sur-Kolay. Optimal Partitionning for FPGA Based Regular Array Implementations. In *IEEE PARELEC'00*, pages 155–159, August 2000. ⟨ p. 56 ⟩

[15] Steven Derrien, Sanjay V. Rajopadhye, and Susmita Sur-Kolay. Combining Instruction and Loop Level Parallelism for Array Synthesis on FPGAs. In *International Symposium on System Synthesis (ISSS'01), Montreal*, pages 273–282, 2001. ⟨ pp. 56, 57 ⟩

[16] Steven Derrien and Tanguy Risset. Interfacing Compiled FPGA Programs: the MMAlpha Approach. In *PDPTA2000: Second International Workshop on Engineering of Reconfigurable Hardware/Software Objects*, pages 189–195. CSREA Press, June 2000. ⟨ p. 56 ⟩

[17] Steven Derrien, Alex Turjan, and Claudiu Zissulescuand Bart Kienhuis. Deriving Efficient Control for Process Networks. In *Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2003. ⟨ p. 64 ⟩

[18] Steven Derrien, Alexandru Turjan, Claudiu Zissulescu, Bart Kienhuis, and Ed F. Deprettere. Deriving efficient control in Process Networks with Compaan/Laura. *International Journal of Embedded Systems*, 3(3):170–180, 2008. ⟨ pp. 7, 57, 64 ⟩

[19] Antoine Floch, Tomofumi Yuki, Clement Guy, Steven Derrien, Benoit Combemale, Sanjay Rajopadhye, and Robert B. France. Model-Driven Engineering and Optimizing Compilers: A bridge too far ? In *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models'11)*, pages 608–622, October 2011. ⟨ p. 74 ⟩

[20] Stéphane Guyetant, Mathieu Giraud, Ludovic L'Hours, Steven Derrien, Stéphane Rubini, Dominique Lavenier, and Frédéric Raimbault. Cluster of Re-configurable Nodes for Scanning Large Genomic Banks. *Parallel Computing*, 31(1):73–96, 2005. ⟨ p. 25 ⟩

[21] Dominique Lavenier, Stéphane Guyetant, Steven Derrien, and Stéphane Rubini. A Reconfigurable Parallel Disk System for Filtering Genomic Banks. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, June 23 - 26, 2003, Las Vegas, Nevada, USA*, pages 154–166, 2003. ⟨ p. 25 ⟩

[22] Antoine Morvan, Steven Derrien, and Patrice Quinton. Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion. In *International Conference on Field-Programmable Technology (FPT'11)*, Beijing, China, 2011. To appear in December 2011. ⟨ p. 66 ⟩

[23] Auguste Noumsi, Steven Derrien, and Patrice Quinton. Acceleration of a Content-Based Image-Retrieval application on the RDISK cluster. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, pages 109–109, 2006. ⟨ pp. 19, 20, 25 ⟩

[24] Muhammad Adeel Pasha, Steven Derrien, and Olivier Sentieys. Ultra Low-Power FSM for Controlled Oriented Applications. In *ISCAS '09: IEEE International Symposium on Circuits and Systems 2009*, pages 1577 – 1580, May 2009. ⟨ pp. 51, 52 ⟩

[25] Muhammad Adeel Pasha, Steven Derrien, and Olivier Sentieys. A Complete Design-Flow for the Generation of Ultra Low-Power WSN Node Architectures Based on Micro-Tasking. In *Proc. of the IEEE/ACM Design Automation Conference (DAC)*, pages 693 –698, Anaheim, CA, USA, June 2010. ⟨ pp. 51, 52 ⟩

[26] Muhammad Adeel Pasha, Steven Derrien, and Olivier Sentieys. System-Level Synthesis for Ultra Low-Power Wireless Sensor Nodes. In *Proc. of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 493–500, Lille, France, September 2010. ⟨ pp. 51, 52 ⟩

[27] Muhammad Adeel Pasha, Steven Derrien, and Olivier Sentieys. System Level Synthesis for Wireless Sensor Node Controllers: A Complete Design Flow. *ACM Transactions on Design Automation of Electronic Systems*, 2011. Accepted for publication on July the 19th 2011. ⟨ pp. 44, 51, 52 ⟩

[28] Muhammad Adeel Pasha, Steven Derrien, and Olivier Sentieys. Toward Ultra Low-Power Hardware Specialization of a Wireless Sensor Network Node. In *INMIC 2009. IEEE International Multi Topic Conference, 2009*, pages 1 – 6, submitted. ⟨ p. 51 ⟩

[29] Rayan Chikhi and Steven Derrien and Auguste Noumsi and Patrice Quinton. Combining Flash Memory and FPGAs to Efficiently Implement a Massively Parallel Algorithm for Content-Based Image Retrieval. *International Journal of Electronics*, 95(7):621 – 635, July 2008. ⟨ pp. 19, 25 ⟩

Appendix D

General references

[30] Autoesl design technologies. http://www.autoesl.com/. ⟨ pp. 6, 56 ⟩

[31] Frama-C, a Framework for Modular Analysis of C programs. http://frama-c.com/download.html. ⟨ p. 72 ⟩

[32] Impulse codeveloper. http://www.impulseaccelerated.com/. ⟨ p. 56 ⟩

[33] The Netezza FAST Engines Framework. http://www.netezza.com/documents/whitepapers/fastengines.pd 2009. ⟨ p. 24 ⟩

[34] *Automatic Generation of FPGA-Specific Pipelined Accelerators*, Mars 2011. ⟨ p. 69 ⟩

[35] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+cl@k: an implementation of lattice-based array contraction in the source-to-source translator rose. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07), San Diego, California, USA, June 13-15, 2007*, pages 73–82, 2007. ⟨ p. 53 ⟩

[36] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: a New Generation of Protein Database Search Programs. *Nucleic Acids Research*, pages 3899–3402, 1997. ⟨ pp. 7, 8, 27 ⟩

[37] Laurent Amsaleg and Patrick Gros. Content-based retrieval using local descriptors: problems and issues from a database perspective. *Pattern Analysis and Applications*, 4(2/3):108–124, 2001. ⟨ pp. 19, 20 ⟩

[38] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM. ⟨ p. 73 ⟩

[39] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing, pages 23-30, Ljubljana*, 2003. ⟨ p. 53 ⟩

[40] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004. ⟨ pp. 53, 66 ⟩

[41] Rodolfo Bezerra Batista, Azzedine Boukerche, and Alba Cristina Magalhaes Alves de Melo. A parallel strategy for biological sequence alignment in restricted memory space. *J. Parallel Distrib. Comput.*, 68:548–561, April 2008. ⟨ p. 38 ⟩

[42] M. Bednara and J. Teich. Interface synthesis for FPGA-based VLSI processor arrays. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA-02)*, Las Vegas, Nevada, U.S.A., 2002. ⟨ p. 57 ⟩

[43] M. W. Benabderrahmane, L-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010. ⟨ p. 53 ⟩

[44] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. PLuTo: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM*

SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, June 2008. ACM. ⟨ pp. 53, 72 ⟩

[45] J.L. Bosque, O.D. Robles, A. Rodriguez, and L. Pastor. Study of a Parallel CBIR Implementation using MPI. In *IEEE International Workshop on Computer Architectures for Machine Perception (CAMP'00)*, 2000. ⟨ p. 19 ⟩

[46] Pierre Boulet and Paul Feautrier. Scanning Polyhedra without Do-loops. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 4, Washington, DC, USA, 1998. IEEE Computer Society. ⟨ pp. 64, 66, 68 ⟩

[47] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260 –264, march 1982. ⟨ p. 35 ⟩

[48] J. Bu, E.F. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In S.Y. Kung and E. Swartzlander, editors, *International Conference on application Specific Array Processing*, pages 591–602, Princeton, New Jersey, September 1990. IEEE Computer Society. ⟨ p. 58 ⟩

[49] Yu Cai, Erich F. Haratsch, Mark McCartney, and Ken Mai. Fpga-based solid-state drive prototyping platform. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '11, pages 101–104, Washington, DC, USA, 2011. IEEE Computer Society. ⟨ p. 24 ⟩

[50] Philippe Clauss and Vincent Loechner. Parametric Analysis of Polyhedral Iteration Spaces. *VLSI Signal Processing*, 19(2):179–194, 1998. ⟨ p. 64 ⟩

[51] J. F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 92–101. ACM, 1995. ⟨ p. 53 ⟩

[52] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *Integration, The VLSI Journal*, 12:293–304, December 1991. ⟨ pp. 57, 58 ⟩

[53] A. Darte and J.-M. Delosme. Partitioning for array processors. Technical Report 90-23, LIP, École normale supérieure de Lyon, France, 1991. ⟨ p. 57 ⟩

[54] A. Darte, R. Schreiber, B. Ramakrishna Rau, and F. Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):159–172, 2002. ⟨ p. 58 ⟩

[55] Alain Darte, Rob Schreiber, Bob Ramakrishna Rau, and Frédéric Vivien. Constructing and Exploiting Linear Schedules with Prescribed Parallelism. *ACM Transactions on Design Automation of Electronic Systems*, 7(1):159–172, 2002. ⟨ p. 57 ⟩

[56] Florent de Dinechin, Jérémie Detrey, Octavian Cret, and Radu Tudoran. When FPGAs are better at floating-point than microprocessors. In *Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays*, page 260, 2008. ⟨ p. 19 ⟩

[57] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. Generating high-performance custom floating-point pipelines. In *19th International Conference on Field Programmable Logic and Applications*, pages 59–64, 2009. ⟨ p. 19 ⟩

[58] B Dutertre and L De Moura. Yices: An SMT Solver. http://yices.csl.sri.com/, 2006. ⟨ p. 72 ⟩

[59] Hritam Dutta, Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. Efficient Control Generation for Mapping Nested Loop Programs onto Processor Arrays. *Journal of Systems Architecture*, 53(5–6):300–309, May 2007. ⟨ p. 64 ⟩

[60] Hritam Dutta, Frank Hannig, and Jürgen Teich. PARO – A Design Tool for Synthesis of Hardware Accelerators for SoCs. Tool Presentation at the University Booth at Design, Automation and Test in Europe (DATE), Dresden, Germany, March 2010. ⟨ p. 54 ⟩

[61] Sean Eddy. HMMER3: a new generation of sequence homology search software. `http://hmmer.janelia.org/`. ⟨ p. 28 ⟩

[62] Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998. ⟨ p. 27 ⟩

[63] Sean R. Eddy. Accelerated profile HMM searches (preprint), 2011. ⟨ p. 6 ⟩

[64] Virantha Ekanayake, Clinton Kelly, IV, and Rajit Manohar. Anultra low-power processor for sensor networks. *SIGOPS Oper. Syst. Rev.*, 38(5):27–36, 2004. ⟨ p. 42 ⟩

[65] Juan Fernando Eusse Giraldo, Nahri Moreano, Ricardo Pezzuol Jacobi, and Alba Cristina Magalhaes Alves de Melo. A hmmer hardware accelerator using divergences. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 405–410, 2010. ⟨ p. 38 ⟩

[66] Mohammed Fellahi and Albert Cohen. Software Pipelining in Nested Loops with Prolog-Epilog Merging. In *High Performance Embedded Architectures and Compilers, Fourth International Conference, HiPEAC 2009, Paphos, Cyprus, January 25-28, 2009. Proceedings*, pages 80–94, 2009. ⟨ p. 69 ⟩

[67] Dirk Fimmel. Generation of Scheduling Functions Supporting LSGP-Partitioning. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'00)*, pages 349–358, Boston, July 2000. ⟨ p. 57 ⟩

[68] A. Fraboulet and T. Risset. Efficient on-chip communications for data-flow IPs. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'04)*, pages 293–303, 2004. ⟨ p. 57 ⟩

[69] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Not.*, 27(4):68–76, 1992. ⟨ p. 45 ⟩

[70] Narayan Ganesan, Roger D. Chamberlain, Jeremy Buhler, and Michela Taufer. Accelerating hmmer on gpus by implementing hybrid data and task parallelism. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, BCB '10, pages 418–421, New York, NY, USA, 2010. ACM. ⟨ pp. 37, 38 ⟩

[71] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006. ⟨ p. 53 ⟩

[72] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *13th International Static Analysis Symposium, SAS'06*, Seoul, Korea, August 2006. ⟨ p. 72 ⟩

[73] M Graphics. Catapult-C Synthesis. http://www.mentor.com. ⟨ pp. 6, 56 ⟩

[74] A.-C. Guillou, P. Quinton, T. Risset, C. Wagner, and D. Massicotte. High-level design of digital filters in mobile communications. DATE Design Contest 2001, March 2001. 2nd place, available at `http://www.irisa.fr/bibli/publi/pi/2001/1405/1405.html`. ⟨ p. 54 ⟩

[75] Anne-Claire Guillou, Patrice Quinton, and Tanguy Risset. Hardware Synthesis for Multi-Dimensional Time. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'03)*, pages 40–50, 2003. ⟨ p. 64 ⟩

[76] T. Han and D. A. Carlson. Fast area-efficient vlsi adders. In *Proceedings of the 8th Symposium on Computer Arithmetic*, pages 49–55. IEEE, 1987. ⟨ p. 35 ⟩

[77] Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. The PAULA Language for Designing Multi-Dimensional Dataflow-Intensive Applications. In *Proceedings of the GI/ITG/GMM-Workshop – Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 129–138, Freiburg, Germany, March 2008. Shaker. ⟨ p. 54 ⟩

[78] Frank Hannig and Jürgen Teich. Output serialization for fpga-based and coarse-grained processor arrays. In *Proceedings of The 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2005, Las Vegas, Nevada, USA*, pages 78–84, 2005. ⟨ p. 60 ⟩

[79] D. Harris. A taxonomy of parallel prefix networks. In *Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 2213 – 2217 Vol.2, nov. 2003. ⟨ p. 35 ⟩

[80] Mark Hempstead, Gu-Yeon Wei, and David Brooks. An Accelerator-Based Wireless Sensor Network Processor in 130nm CMOS. In *CASES'09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 215–222, New York, NY, USA, 2009. ACM. ⟨ p. 50 ⟩

[81] Rob Hess. An open-source siftlibrary. In *Proceedings of the international conference on Multimedia*, MM '10, pages 1493–1496, New York, NY, USA, 2010. ACM. ⟨ p. 25 ⟩

[82] D.T. Hoang. Searching genetic databases on splash 2. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pages 185 –191, apr 1993. ⟨ p. 13 ⟩

[83] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. In *SC'05 : Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005. ⟨ p. 37 ⟩

[84] INRIA. PowWow, Protocol for Low Power Wireless Sensor Network, http://powwow.gforge.inria.fr/. ⟨ p. 47 ⟩

[85] F. Irigoin and R. Triolet. Supernode partitioning. In 15*th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988. ⟨ p. 57 ⟩

[86] Gilles Kahn. The Semantics of a Simple Language For Parallel Programming. In *Proceedings of the IFIP Congress 74*. North-Holland Publishing Co., 1974. ⟨ p. 56 ⟩

[87] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, 1998. ⟨ pp. 7, 14 ⟩

[88] A.C.J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Method and Tools*. PhD thesis, Delft University of Technology, January 1999. ⟨ p. 63 ⟩

[89] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, USA, May 2000. ⟨ p. 61 ⟩

[90] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Deriving Process Networks from Nested Loop Alogorithms. In *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000. ⟨ p. 56 ⟩

[91] S. Knowles. A family of adders. In *ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, page 30, Washington, DC, USA, 1999. IEEE Computer Society. ⟨ p. 35 ⟩

[92] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transcation on Computers*, 22(8):786–793, 1973. ⟨ p. 35 ⟩

[93] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov Models in Computational Biology: Applications to Protein Modeling. *Journal Molecular Biology*, 235:1501–1531, February 1994. ⟨ p. 27 ⟩

[94] H T Kung and Charles E Leiserson. *Algorithms for VLSI Processor Arrays.* Addison Wesley, 1978. ⟨ p. 54 ⟩

[95] J. Kwong, Y.K. Ramadass, N. Verma, and A.P. Chandrakasan. A 65 nm Sub-Vt Microcontroller With Integrated SRAM and Switched Capacitor DC-DC Converter. *IEEE Journal of Solid-State Circuits*, 44(1):115 –126, jan. 2009. ⟨ p. 42 ⟩

[96] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of ACM*, 27(4):831–838, 1980. ⟨ p. 35 ⟩

[97] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM. ⟨ p. 69 ⟩

[98] D. Lavenier, G. Georges, and X. Liu. A reconfigurable index flash memory tailored to seed-based genomic sequence comparison algorithms. *J. VLSI Signal Process. Syst.*, 48:255–269, September 2007. ⟨ p. 18 ⟩

[99] Herwig Lejsek, Björn ■ór Jónsson, and Laurent Amsaleg. Nv-tree: nearest neighbors at the billion scale. In *Proceedings of the 1st ACM International Conference on Multimedia Retrieval*, ICMR '11, pages 54:1–54:8, New York, NY, USA, 2011. ACM. ⟨ p. 25 ⟩

[100] Ludovic L'Hours. Generating efficient custom fpga soft-cores for control-dominated applications. In *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2005), 23-25 July 2005, Samos, Greece*, pages 127–133, 2005. ⟨ p. 41 ⟩

[101] Teng Li, Miaoqing Huang, Tarek El-Ghazawi, and H. Howie Huang. Reconfigurable Active Disk: An FPGA Accelerated Storage Architecture for Data-Intensive Applications. In *Symposium on Application Accelerators in High-Performance Computing (SAAHPC'09)*, 2009. ⟨ p. 24 ⟩

[102] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD'95*, pages 393–399, Nov 1995. ⟨ p. 45 ⟩

[103] E.-Y.A. Lin, J.M. Rabaey, and A. Wolisz. Power-efficient rendez-vous schemes for dense wireless sensor networks. In *Communications, 2004 IEEE International Conference on*, volume 7, pages 3769–3776 Vol.7, June 2004. ⟨ p. 43 ⟩

[104] Eric Lindahl. HMMer Altivec Implementation. http://lindahl.sbc.su.se/software/altivec/altivec-hmmer,-version-2.html, 2005. ⟨ p. 37 ⟩

[105] L.L'Hours. Generating Efficient Custom FPGA Soft-Cores for Control-Dominated Applications. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture Processors: ASAP '05*, pages 127–133, Washington, DC, USA, 2005. IEEE Computer Society. ⟨ pp. 9, 45 ⟩

[106] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 133–142, New York, NY, USA, 2009. ACM. ⟨ p. 24 ⟩

[107] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin, and Brandon Harris. Accelerator Design for Protein Sequence HMM Search. In *Proceedings of the ACM International Conference on Supercomputing*, Cairns, Australia, 2006. ACM. ⟨ pp. 37, 38 ⟩

[108] Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch, and Francois Charot. Constraint-Driven Instructions Selection and Application Scheduling in the DURASE System. *IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP'09*. ⟨ p. 45 ⟩

[109] Sjoerd Meijer, Hristo Nikolov, and Todor Stefanov. On compile-time evaluation of process partitioning transformations for kahn process networks. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 31–40, New York, NY, USA, 2009. ACM. ⟨ p. 56 ⟩

[110] Gokhan Memik, Mahmut T. Kandemir, and Alok Choudhary. Design and evaluation of smart disk cluster for dss commercial workloads. *Journal of Parallel and Distributed Computing (JPDC)*, 61(11):1633–1664, 2001. ⟨ pp. 7, 14 ⟩

[111] Dan I Moldovan and Jose A. B Fortes. Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays. *IEEE Transactons on Computers*, 35(1):1–12, 1986. ⟨ p. 58 ⟩

[112] D.I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Transactions on Computers*, 35(1):1–12, jan 1986. ⟨ p. 57 ⟩

[113] Rene Mueller and Jens Teubner. Fpgas: a new point in the database design space. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 721–723, New York, NY, USA, 2010. ACM. ⟨ p. 24 ⟩

[114] Kalyan Muthukumar and Gautam Doshi. Software pipelining of nested loops. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 165–181, London, UK, 2001. Springer-Verlag. ⟨ p. 65 ⟩

[115] S. Mysore, B. Agrawal, F.T. Chong, and T. Sherwood. Exploring the Processor and ISA Design for Wireless Sensor Network Applications. In *21st International Conference on VLSI Design, 2008. VLSI'08*, pages 59–64, Jan. 2008. ⟨ p. 47 ⟩

[116] L. Nazhandali, M. Minuth, and T. Austin. SenseBench: Toward an Accurate Evaluation of Sensor Network Processors. In *Proceedings of the IEE International Workload Characterization Symposium, 2005*, pages 197–203, Oct. 2005. ⟨ p. 47 ⟩

[117] T. Oliver, B. Schmidt, Y. Jakop, and D. L. Maskell. Accelerating the Viterbi Algorithm for Profile Hidden Markov Models Using Reconfigurable Hardware. In *International Conference on Computational Science*, 2006. ⟨ pp. 33, 37, 38 ⟩

[118] T. Oliver, L. Y. Yeow, and B. Schmidt. High Performance Database Searching with HMMer on FPGAs. In *HiCOMB 2007, Sixth IEEE International Workshop on High Performance Computational Biology*, march 2007. ⟨ pp. 37, 38 ⟩

[119] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks* . Book Chapter in Ambient Intelligence by Springer, 2005. ⟨ p. 42 ⟩

[120] Joonseok Park and Pedro C. Diniz. Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines. In *International Symposium on System Synthesis (ISSS'01)*, pages 221–226, 2001. ⟨ p. 57 ⟩

[121] Joonseok Park and Pedro C. Diniz. Synthesis and estimation of memory interfaces for FPGA-based reconfigurable computing engines. In *International Symposium on FPGA Custom Computing Machines*, 2003. ⟨ p. 57 ⟩

[122] Fabien A.P. Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Attacks on copyright marking systems. In *Information Hiding*, pages 218–238, 1998. ⟨ pp. 18, 20 ⟩

[123] Alexandru Plesco. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*. PhD thesis, Ecole normale supÃl'rieure de Lyon, 2010. ⟨ pp. 56, 60, 72 ⟩

[124] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press. ⟨ pp. 53, 73 ⟩

[125] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop Transformations: Convexity, Pruning and Optimization . In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, pages 549–562, Austin, TX, January 2011. ACM Press. ⟨ p. 53 ⟩

[126] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 208–214, New York, NY, USA, 1984. ACM. ⟨ pp. 30, 54 ⟩

[127] L. Rabiner and B. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4 – 16, jan 1986. ⟨ p. 27 ⟩

[128] Raval, R. K. et al. Low-Power TinyOS Tuned Processor Platform for Wireless Sensor Network Motes. *ACM Transactions on Design Automation of Electronic Systems*, 15(3):1–17, 2010. ⟨ p. 42 ⟩

[129] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 405–414, New York, NY, USA, 2007. ACM. ⟨ p. 73 ⟩

[130] CAIRN-INRIA research group. The gecos: The generic compiler suite. `http://gecos.gforge.inria.fr/`. ⟨ p. 56 ⟩

[131] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, june 2001. ⟨ pp. 7, 14 ⟩

[132] Edwin Rijpkema. *From Piecewise Regular Algorithms to Dataflow Architectures*. PhD thesis, Delft University of Technology, 2001. ⟨ p. 61 ⟩

[133] Oscar D. Robles, Josï£¡ L. Bosque, Luis Pastor, and Angel Rodrï£¡guez. Performance Analysis of a CBIR System on Shared-Memory Systems and Heterogeneous Clusters. In *IEEE International Workshop on Computer Architectures for Machine Perception (CAMP'05)*, 2005. ⟨ p. 19 ⟩

[134] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multidimensional loops. *ACM Trans. Archit. Code Optim.*, 4, March 2007. ⟨ p. 69 ⟩

[135] Robert Schreiber, Shail Aditya, Bob Ramakrisha Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-level synthesis of non programmable hardware accelerators. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'00)*, pages 113–126, Boston, July 2000. ⟨ pp. 57, 58 ⟩

[136] Seok, Mingoo et al. The Phoenix Processor: A 30pW Platform for Sensor Applications. In *VLSI'08: Proceedings of the IEEE Symposium on VLSI Circuits*, pages 188–189, 2008. ⟨ p. 42 ⟩

[137] W. Shang and J. A. B. Fortes. Independent partitioning of algorithms with uniform dependencies. In *International Conference on Parallel Processing (ICPP'88)*, pages 26–33, 1988. ⟨ p. 57 ⟩

[138] M. Sheets, F. Burghardt, T. Karalar, J. Ammer, Y.H. Chee, and J. Rabaey. A power-managed protocol processor for wireless sensor networks. In *Symposium on VLSI Circuits, 2006. Digest of Technical Papers*, pages 212 –213, 0-0 2006. ⟨ p. 42 ⟩

[139] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226 –231, june 1960. ⟨ p. 35 ⟩

[140] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol*, 147:195–197, 1981. ⟨ pp. 8, 27 ⟩

[141] Clinton Wills Smullen, IV, Shahrukh Rohinton Tarapore, Sudhanva Gurumurthi, Parthasarathy Ranganathan, and Mustafa Uysal. Active storage revisited: the case for power and performance benefits for unstructured data processing applications. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 293–304, 2008. ⟨ p. 24 ⟩

[142] Pascal Sotin and Bertrand Jeannet. Precise Interprocedural Analysis in the Presence of Pointers to the Stack. ⟨ p. 72 ⟩

[143] Jim Steel and Jean-Marc Jézéquel. Model Typing for Improving Reuse in Model-Driven Engineering. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, LNCS, pages –, Berlin, Heidelberg, 2005. Springer-Verlag. ⟨ p. 75 ⟩

[144] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 7–12, New York, NY, USA, 2002. ACM. ⟨ p. 56 ⟩

[145] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks: The compaan/laura approach. In *Proceedings of DATE2004*, Paris, France, Feb 16 – 20 2004. ⟨ p. 61 ⟩

[146] Yanteng Sun, Peng Li, Guochang Gu, Yuan Wen, Yuan Liu, , and Dong Liu. HMMER Acceleration Using Systolic Array Based Reconfigurable Architecture. In *IEEE International Workshop on High Performance Computational Biology*, 2009. ⟨ pp. 37, 38 ⟩

[147] Toyokazu Takagi and Tsutomu Maruyama. Accelerating HMMER Search Using FPGA. In *International Conference on Field Programmable Logic and Applications*, September 2009. ⟨ pp. 37, 38 ⟩

[148] Sanket Tavarageri, Louis-Noël Pouchet, J. Ramanujam andAtanas Rountev, and P. Sadayappan. Dynamic Selection of Tile Sizes . In *18th annual IEEE International Conference on High Performance Computing (HiPC'11)*, Bangalore, India, December 2011. IEEE Computer Society Press. ⟨ p. 73 ⟩

[149] Jürgen Teich, Lothar Thiele, and Lee Z. Zhang. Partitioning Processor Arrays under Resource Constraints. *VLSI Signal Processing*, 17(1):5–20, 1997. ⟨ p. 57 ⟩

[150] Texas Instruments. MSP430 User Guide. Technical report, 2006. ⟨ pp. 9, 41 ⟩

[151] The Eclipse Foundation. Xtext, a framework for development of textual domain specific languages (DSLs), http://www.eclipse.org/Xtext/. ⟨ p. 47 ⟩

[152] Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. Classifying interprocess communication in process network representation of nested-loop programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2), 2007. ⟨ p. 56 ⟩

[153] Alexandru Turjan, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. The Compaan Tool Chain: Converting Matlab into Process Networks. In *In Designers' Forum "Design, Automation and Test in Europe (DATE 2002)", Paris, France, 2002.* ⟨ pp. 7, 56, 57 ⟩

[154] Sven Verdoolaege. *Handbook of Signal Processing Systems*, chapter Polyhedral process networks. Springer, Heidelberg, Germany, 1 edition, 2004. ⟨ pp. 7, 56, 57, 61 ⟩

[155] Emmanuel Viaud, François Pêcheux, and Alain Greiner. An efficient tlm/t modeling and simulation environment based on conservative parallel discrete event principles. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 94–99, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. ⟨ p. 24 ⟩

[156] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systemes*, 4(1), March 1996. ⟨ p. 13 ⟩

[157] Jean E. Vuillemin. On computing power. pages 69–86, 1994. ⟨ p. 13 ⟩

[158] J. P. Walters, B. Qudah, and V. Chaudhary. Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, 2006. ⟨ p. 37 ⟩

[159] John Paul Walters, Vidyananth Balu, Suryaprakash Kompalli, and Vipin Chaudhary. Evaluating the use of gpus in liver image segmentation and hmmer database searches. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. ⟨ p. 37 ⟩

[160] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: a power-performance simulator for interconnection networks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 294–305, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ⟨ p. 24 ⟩

[161] D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993. ⟨ p. 54 ⟩

[162] Ben Wun, Jeremy Buhler, and Patrick Crowley. Exploiting Coarse-Grained Parallelism to Accelerate Protein Motif Finding with a Network Processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005. ⟨ p. 37 ⟩

[163] Jingling Xue and Christian Lengauer. The Synthesis of Control Signals for One-Dimensional Systolic Arrays. *Integration*, 14:1–32, 1992. ⟨ p. 54 ⟩

[164] Claudiu Zissulescu, Bart Kienhuis, and Ed F. Deprettere. Expression Synthesis in Process Networks generated by LAURA. In *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2005)*, pages 15–21, 2005. ⟨ p. 64 ⟩

[165] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed F. Deprettere. Laura: Leiden Architecture Research and Exploration Tool. In *Field Programmable Logic and Applications (FPL'03)*, pages 911–920, 2003. ⟨ pp. 7, 57 ⟩