INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team compsys

Compilation and Embedded Computing Systems

Grenoble - Rhône-Alpes

Theme : Architecture and Compiling

Table of contents

1. Team **1**
2. Overall Objectives **1**
2.1. Introduction 1
2.2. General presentation 2
2.3. Highlights of the first 4-years period 4
2.4. Highlights of 2010 4
3. Scientific Foundations **5**
3.1. Introduction 5
3.2. Back-end code optimizations for embedded processors 7
3.2.1. Embedded systems and the revival of compilation & code optimizations 7
3.2.2. Aggressive and just-in-time optimizations of assembly-level code 8
3.3. Program analysis and transformations for high-level synthesis 9
3.3.1. High-Level Synthesis Context 9
3.3.2. Specifications, Transformations, Code Generation for High-Level Synthesis 10
4. Application Domains **12**
5. Software **12**
5.1. Introduction 12
5.2. Pip 13
5.3. Syntol 13
5.4. Cl@k 14
5.5. Bee 14
5.6. Chuba 15
5.7. RanK 15
5.8. C2fsm 16
5.9. LAO developments in aggressive compilation 16
5.10. LAO developments in JIT compilation 16
6. New Results **16**
6.1. Introduction 16
6.2. Split register allocation: linear complexity without performance penalty 17
6.3. Parallel copy motion and critical edge splitting 17
6.4. Static single information form: debunking 18
6.5. “Optimal” formulation of register spilling 18
6.6. Fast computation of liveness sets 18
6.7. Decoupled graph-coloring register allocation with hierarchical aliasing 19
6.8. Graph-coloring and tree-scan register allocation using repairing 19
6.9. Program analysis and communication optimizations for HLS 20
6.10. Loop transformations for pipelined floating-point arithmetic operators 20
6.11. Program termination and worst-case computational complexity 21
6.12. Completeness of instruction selectors 21
6.13. Execution models for processors and instructions 22
7. Contracts and Grants with Industry **23**
7.1. Nano2012 MEDIACOM project with stmicroelectronics on SSA, register allocation, and JIT compilation 23
7.2. Nano2012 S2S4HLS project with stmicroelectronics on source-to-source transformations for high-level synthesis 23
8. Other Grants and Activities **23**
8.1. National initiatives 23
8.2. European initiatives 23
8.3. International initiatives 24

8.4. Informal cooperations 24
9. Dissemination **24**
9.1. Conferences, journals, and book chapters 24
9.2. Teaching and thesis advising 25
9.3. PhD defense committees and hiring committees 25
9.4. Workshops, seminars, and invited talks 25
10. Bibliography **26**

The objective of Compsys is to adapt and extend optimization techniques, primarily designed for high performance computing, to the special case of embedded computing systems. The team exists since January 2002 as part of Laboratoire de l'Informatique du Parallélisme (Lip, UMR CNRS ENS-LYON UCB-LYON *Inria 5668), located at* ENS-LYON*, and as an Inria pre-project. It became a full Inria project in January 2004. It has been evaluated by Inria in Spring 2007 and will continue 4 more years. It has been evaluated by* AERES *in December 2010 and received the mark A+.*

1. Team

Research Scientists

Christophe Alias [Junior Researcher (CR) Inria, Jan. 2009-...]
Alain Darte [Senior Researcher (DR) CNRS, Team Leader, HdR]
Fabrice Rastello [Junior Researcher (CR) Inria]

Faculty Members

Paul Feautrier [Professor ENS-LYON, emeritus, HdR]
Laure Gonnord [Associate Professor (Lille University), external collaborator, part-time]

PhD Students

Benoit Boissinot [ENS-LYON grant, Sept. 2006-Sept. 2010]
Quentin Colombet [inria, contract Nano2012 Mediacom, Jan. 2010-...]
Alexandru Plesco [MESR grant, then 1/2 ATER INSA, then CNRS grant, Sept. 2006-Dec. 2010]

Post-Doctoral Fellow

Florian Brandner [Inria, contract Nano2012 Mediacom, Dec. 2009-Dec. 2010]

Administrative Assistants

Caroline Suter [Inria, part-time]
Laetitia Lecot [Inria, part-time]

2. Overall Objectives

2.1. Introduction

Keywords: compilation, automatic generation of VLSI chips, code optimization, scheduling, parallelism, memory optimization, FPGA platforms, VLIW processors, DSP, regular computations, linear programming, tools for polyhedra and lattices.

The objective of Compsys is to adapt and to extend code optimization techniques primarily designed in compilers/parallelizers for high performance computing to the special case of *embedded computing systems*. In particular, Compsys works on back-end optimizations for specialized processors and on high-level program transformations for the synthesis of hardware accelerators. The main characteristic of Compsys is its focus on combinatorial problems (graph algorithms, linear programming, polyhedra) coming from code optimizations (register allocation, cache and memory optimizations, scheduling, optimizations for power, automatic generation of software/hardware interfaces, etc.) and the validation of techniques developed in compilation tools.

Compsys started as an Inria project in 2004, after 2 years of maturation, and was positively evaluated in Spring 2007 after its first 4 years period (2004-2007). It was again evaluated by AERES in 2009, as part of the general evaluation of LIP, and got the best possible mark, A+. It will continue with updated research directions. Section 2.2 defines the general context of the team's activities. Section 2.3 presents the research objectives targeted during the first 4 years, the main achievements over this period, and the new research directions that Compsys will follow in the coming years. The last section highlights new results that have not been covered by previous reports.

2.2. General presentation

Classically, an embedded computer is a digital system that is part of a larger system and that is not directly accessible to the user. Examples are appliances like phones, TV sets, washing machines, game platforms, or even larger systems like radars and sonars. In particular, this computer is not programmable in the usual way. Its program, if it exists, is supplied as part of the manufacturing process and is seldom (or never) modified thereafter. As the embedded systems market grows and evolves, this view of embedded systems is becoming obsolete and tends to be too restrictive. Many aspects of general-purpose computers apply to modern embedded platforms. Nevertheless, embedded systems remain characterized by a set of specialized application domains, rigid constraints (cost, power, efficiency, heterogeneity), and its market structure. The term *embedded system* has been used for naming a wide variety of objects. More precisely, there are two categories of so-called *embedded systems*: a) control-oriented and hard real-time embedded systems (automotive, plant control, airplanes, etc.); b) compute-intensive embedded systems (signal processing, multi-media, stream processing) processing large data sets with parallel and/or pipelined execution. Compsys is primarily concerned with this second type of embedded systems, now referred to as *embedded computing systems*.

Today, the industry sells many more embedded processors than general-purpose processors; the field of embedded systems is one of the few segments of the computer market where the European industry still has a substantial share, hence the importance of embedded system research in the European research initiatives. Our priority towards embedded software is motivated by the following observations: a) the embedded system market is expanding, among many factors, one can quote pervasive digitalization, low-cost products, appliances, etc.; b) research on software for embedded systems is poorly developed in France, especially if one considers the importance of actors like Alcatel, STMicroelectronics, Matra, Thales, etc.; c) since embedded systems increase in complexity, new problems are emerging: computer-aided design, shorter time-to-market, better reliability, modular design, and component reuse.

A specific aspect of embedded computing systems is the use of various kinds of processors, with many particularities (instruction sets, registers, data and instruction caches) and constraints (code size, performance, storage). The development of *compilers* is crucial for this industry, as selling a platform without its programming environment and compiler would not be acceptable. To cope with such a range of different processors, the development of robust, generic (retargetable), though efficient compilers is mandatory. Unlike standard compilers for general-purpose processors, compilers for embedded processors can be more aggressive (i.e., take more time to optimize) for optimizing some important parts of applications. This opens a new range of optimizations. Another interesting aspect is the introduction of platform-independent intermediate languages, such as Java bytecode, that is compiled dynamically at runtime (aka just-in-time). Extreme lightweight compilation mechanisms that run faster and consume less memory have to be developed. Our objective is to revisit existing compilation techniques in the context of embedded computing systems, to deconstruct these techniques, to improve them, and to develop new techniques taking constraints of embedded processors into account.

As for *high-level synthesis* (HLS), several compilers/systems have appeared, after some first unsuccessful industrial attempts in the past. These tools are mostly based on C or C++ as for example SystemC, VCC, CatapultC, Altera C2H, PICO Express. Academic projects also exist such as Flex and Raw at MIT, Piperench at Carnegie-Mellon University, Compaan at the University of Leiden, Ugh/Disydent at LIP6 (Paris), Gaut at Lester (Bretagne), MMAlpha (Insa-Lyon), and others. In general, the support for parallelism in HLS tools is minimal, especially in industrial tools. Also, the basic problem that these projects have to face is that the definition of performance is more complex than in classical systems. In fact, it is a multi-criteria optimization problem and one has to take into account the execution time, the size of the program, the size of the data structures, the power consumption, the manufacturing cost, etc. The impact of the compiler on these costs is difficult to assess and control. Success will be the consequence of a detailed knowledge of all steps of the design process, from a high-level specification to the chip layout. A strong cooperation of the compilation and chip design communities is needed. The main expertise in Compsys for this aspect is in the *parallelization* and optimization of *regular computations*. Hence, we will target applications with a large potential parallelism, but we will attempt to integrate our solutions into the big picture of CAD environments.

More generally, the aims of Compsys are to develop new compilation and optimization techniques for the field of embedded computing system design. This field is large, and Compsys does not intend to cover it in its entirety. As previously mentioned, we are mostly interested in the automatic design of accelerators, for example designing a VLSI or FPGA circuit for a digital filter, and in the development of new back-end compilation strategies for embedded processors. We study code transformations that optimize features such as execution time, power consumption, code and die size, memory constraints, and compiler reliability. These features are related to embedded systems but some are not specific to them. The code transformations we develop are both at source level and at assembly level. A specificity of Compsys is to mix a solid theoretical basis for all code optimizations we introduce with algorithmic/software developments. Within Inria, our project is related to the "architecture and compilation" theme, more precisely code optimization, as some of the research in Alchemy and Alf (previously known as Caps), and to high-level architectural synthesis, as some of the research in Cairn.

Most french researchers working on high-performance computing (automatic parallelization, languages, operating systems, networks) moved to grid computing at the end of the 90s. We thought that applications, industrial needs, and research problems were more interesting in the design of embedded platforms. Furthermore, we were convinced that our expertise on high-level code transformations could be more useful in this field. This is the reason why Tanguy Risset came to Lyon in 2002 to create the Compsys team with Anne Mignotte and Alain Darte, before Paul Feautrier, Antoine Fraboulet, Fabrice Rastello, and finally Christophe Alias joined the group. Then, Tanguy Risset left Compsys to become a professor at INSA Lyon, and Antoine Fraboulet and Anne Mignotte moved to other fields of research. As for Laure Gonnord, after a post-doc in Compsys, she obtained an assistant professor position in Lille but remains external collaborator of the team.

All present and past members of Compsys have a background in automatic parallelization and high-level program transformations. Paul Feautrier was the initiator of the polytope model for program transformations around 1990 and, before coming to Lyon, started to be more interested in programming models and optimizations for embedded applications, in particular through collaborations with Philips. Alain Darte worked on mathematical tools and algorithmic issues for parallelism extraction in programs. He became interested in the automatic generation of hardware accelerators, thanks to his stay at HP Labs in the Pico project in Spring 2001. Antoine Fraboulet did a PhD with Anne Mignotte – who was working on high-level synthesis (HLS) – on code and memory optimizations for embedded applications. Fabrice Rastello did a PhD on tiling transformations for parallel machines, then was hired by STMicroelectronics where he worked on assembly code optimizations for embedded processors. Tanguy Risset worked for a long time on the synthesis of systolic arrays, being the main architect of the HLS tool MMAlpha. Christophe Alias did a PhD on algorithm recognition for program optimizations and parallelization. He first spent a year in Compsys working on array contraction, where he started to develop his tool Bee, then a year at Ohio State University with Prof. P. Sadayappan on memory optimizations. He finally joined Compsys as an Inria researcher.

It may be worth to quote Bob Rau and his colleagues (IEEE Computer, sept. 2002):

"Engineering disciplines tend to go through fairly predictable phases: ad hoc, formal and rigorous, and automation. When the discipline is in its infancy and designers do not yet fully understand its potential problems and solutions, a rich diversity of poorly understood design techniques tends to flourish. As understanding grows, designers sacrifice the flexibility of wild and woolly design for more stylized and restrictive methodologies that have underpinnings in formalism and rigorous theory. Once the formalism and theory mature, the designers can automate the design process. This life cycle has played itself out in disciplines as diverse as PC board and chip layout and routing, machine language parsing, and logic synthesis.

We believe that the computer architecture discipline is ready to enter the automation phase. Although the gratification of inventing brave new architectures will always tempt us, for the most part the focus will shift to the automatic and speedy design of highly customized computer systems using well-understood architecture and compiler technologies."

We share this view of the future of architecture and compilation. Without targeting too ambitious objectives, we were convinced of two complementary facts: a) the mathematical tools developed in the past for manipulating programs in automatic parallelization were lacking in high-level synthesis and embedded computing

optimizations and, even more, they started to be rediscovered frequently in less mature forms, b) before being able to really use these techniques in HLS and embedded program optimizations, we needed to learn a lot from the application side, from the electrical engineering side, and from the embedded architecture side. Our primary goal was thus twofold: to increase our knowledge of embedded computing systems and to adapt/extend code optimization techniques, primarily designed for high performance computing, to the special case of embedded computing systems. In the initial Compsys proposal, we proposed four research directions, centered on compilation methods for embedded applications, both for software and accelerators design:

- Code optimization for specific processors (mainly DSP and VLIW processors);
- Platform-independent loop transformations (including memory optimization);
- Silicon compilation and hardware/software codesign;
- Development of polyhedral (but not only) optimization tools.

These research activities were primarily supported by a marked investment in polyhedra manipulation tools and, more generally, solid mathematical and algorithmic studies, with the aim of constructing operational software tools, not just theoretical results. Hence the fourth research theme was centered on the development of these tools.

2.3. Highlights of the first 4-years period

The Compsys team has been evaluated by Inria in April 2007. The evaluation, conducted by Erik Hagersted (Uppsala University), Vinod Kathail (Synfora, inc), J. (Ram) Ramanujam (Baton Rouge University) was positive. Compsys will thus continue for 4 years as an Inria project-team but in a new configuration as Tanguy Risset and Antoine Fraboulet left the project to follow research directions closer to their host laboratory at Insa-Lyon. The main achievements of Compsys, for this period, were the following:

- The development of a strong collaboration with the compilation group at STMicroelectronics, with important results in aggressive optimizations for instruction cache and register allocation.
- New results on the foundation of high-level program transformations, including scheduling techniques for process networks and a general technique for array contraction (memory reuse) based on the theory of lattices.
- Many original contributions with partners closer to hardware constraints, including CEA, related to SoC simulation, hardware/software interfaces, power models, and simulators.

Due to the size reduction of Compsys (from 5 permanent researchers to 3 in 2008, then 4 again in 2009), the team now focuses on two research directions only:

- Code generation for embedded processors, on the two opposite, though connected, aspects: aggressive compilation and just-in-time compilation.
- High-level program analysis and transformations for high-level synthesis tools.

2.4. Highlights of 2010

Compsys has continued its activities on static single assignment (SSA) and register allocation, in collaboration with STMicroelectronics, but working more deeply on just-in-time compilation (in particular on the developments of code optimizations algorithms that take into account speed and memory footprint) and more specific constraints such as register aliasing. This work has led to four new developments in 2010:

- A debunking work on static single information (SSI), which is an extension of SSA: in particular it provides a correct ordering of basic blocks for which live-ranges of variables form intervals.
- A new algorithm for liveness analysis under SSA and a comparison with existing methods, depending on the data structures used.
- An analysis, based on an integer linear programming formulation, of what “optimal spilling” means and on the impact of SSA on this problem.
- A tree-scan register allocator that takes into account more complex aliasing register constraints.

The Sceptre Minalogic project was finished and (very) positively evaluated at the end of 2009. The collaboration with STMicroelectronics continues, since September 2009, through the Mediacom project, a 3-years project funded by Nano2012 (governmental help for R&D). One PhD thesis was defended on this topic in September 2010 by Benoit Boissinot. Another one started in January 2010 (Quentin Colombet) and a post-doctoral researcher was hired in December 2009 (Florian Brandner). It is also worth mentioning the departure of Fabrice Rastello for a sabbatical year since July 2010.

The research on high-level synthesis (HLS) is mainly funded by a 4-years Nano2012 project in collaboration with the Cairn Inria project and STMicroelectronics, on source-to-source transformations for high-level synthesis (S2S4HLS). On the topic of HLS, three aspects have been pushed in 2010:

- In the context of the industrial HLS tool Altera C2H, an automatic method and a software tool (Chuba) have been developed to optimize communications and overlap communications/computations, entirely at source level, without requiring any additional VHDL glue. This technique is the heart of the PhD of Alexandru Plesco who defended his thesis in September 2010. It uses sophisticated polyhedral program analysis, array contraction, and code rewriting techniques.
- The previous effort opened a collaboration with the Arenaire LIP team ("computer arithmetic") for exploiting floating-point pipelined operators in a HLS flow. A prototype has been developed.
- Our work on program termination and worst-case computational complexity has been acknowledged by the community through a publication at SAS'10 and the visit of Amir Ben Amram (University of Tel-Aviv), in connection with the PLUME LIP team ("proofs and languages"). The initial motivation of this research was to be able to analyze/transform codes with while loops so that they can be accepted by HLS tools. Finally, the most important result is that our technique bridges the gap between several communities: program termination, parallelism detection, and analysis of recurrence equations. Furthermore, we can reuse all polyhedral tools developed in high-performance computing: our multi-dimensional scheduling techniques as well as the mathematical tools for manipulating polyhedra and counting integer points within polyhedra.

3. Scientific Foundations

3.1. Introduction

The embedded system design community is facing two challenges:

- The complexity of embedded applications is increasing at a rapid rate.
- The needed increase in processing power is no longer obtained by increases in the clock frequency, but by increased parallelism.

While, in the past, each type of embedded application was implemented in a separate appliance, the present tendency is toward a universal hand-held object, which must serve as a cell-phone, as a personal digital assistant, as a game console, as a camera, as a Web access point, and much more. One may say that embedded applications are of the same level of complexity as those running on a PC, but they must use a more constrained platform in term of processing power, memory size, and energy consumption. Furthermore, most of them depend on international standards (e.g., in the field of radio digital communication), which are evolving rapidly. Lastly, since ease of use is at a premium for portable devices, these applications must be integrated seamlessly to a degree that is unheard of in standard computers.

All of this dictates that modern embedded systems retain some form of programmability. For increased designer productivity and reduced time-to-market, programming must be done in some high-level language, with appropriate tools for compilation, run-time support, and debugging. This does not mean that all embedded systems (or all of an embedded system) must be processor based. Another solution is the use of field programmable gate arrays (FPGA), which may be programmed at a much finer grain than a processor, although the process of FPGA "programming" is less well understood than software generation. Processors are better

than application-specific circuits at handling complicated control and unexpected events. On the other hand, FPGAs may be tailored to just meet the needs of their application, resulting in better energy and silicon area usage. It is expected that most embedded systems will use a combination of general-purpose processors, specific processors like DSPs, and FPGA accelerators. Such a combination is already present in recent versions of the Atom Intel processor.

Solid state technology has changed pace around the turn of the century. Moore's "law", which dictated a twofold increases of the number of gates per chip every eighteen months, is now taking nearer three years for each new technology node. As the number of insulating atoms in a transistor gate and the number of electrons that make a one bit in memory are getting closer to one, it is to be expected that this slowdown will continue until progress stops. At the same time, while miniaturization of silicon features continues, albeit at a slower pace, it has not been possible to reduce the size of metallic interconnects, especially those that carry the clock signal of synchronous designs. The clock "tree" now accounts for 30 % of the energy budget of a typical chip. Further increases of the clock frequency would raise this number beyond the competence of portable cooling systems. The problem is still more acute for embedded systems, which usually run on battery power. As a consequence, the clock frequency of general-purpose chips is limited to less than 3Ghz. Pending a revolution in solid state technology – one not so remote possibility would be to move into asynchronous, clock-less designs – the extra processing power needed to support modern embedded applications must be found elsewhere, namely in increased parallelism. Traditionally, most state-of-the-art processors have parallelism, but this parallelism is hidden to the programmer, which is presented with a sequential execution model. However, this approach is showing diminishing returns, hence the advent of EPIC and multi- or many-core processors.

As a consequence, parallel programming, which has long been confined to the high-performance community, must become the commonplace rather than the exception. In the same way that sequential programming moved from assembly code to high-level languages at the price of a slight loss in performance, parallel programming must move from low-level tools, like OpenMP or even MPI, to higher-level programming environments. While fully-automatic parallelization is a Holy Grail that will probably never be reached in our lifetimes, it will remain as a component in a comprehensive environment, including general-purpose parallel programming languages, domain-specific parallelizers, parallel libraries and run-time systems, back-end compilation, dynamic parallelization. The landscape of embedded systems is indeed very diverse and many design flows and code optimization techniques must be considered. For example, embedded processors (micro-controllers, DSP, VLIW) require powerful back-end optimizations that can take into account hardware specificities, such as special instructions and particular organizations of registers and memories. FPGA and hardware accelerators, to be used as small components in a larger embedded platform, require "hardware compilation", i.e., design flows and code generation mechanisms to generate non-programmable circuits. For the design of a complete system-on-chip platform, architecture models, simulators, debuggers are required. The same is true for multi-cores of any kind, GPGPU ("general-purpose" graphical processing units), CGRA (coarse-grain reconfigurable architectures), which require specific methodologies and optimizations, although all these techniques converge or have connections. In other words, embedded systems need all usual aspects of the process that transforms some specification down to an executable, software or hardware. In this wide range of topics, Compsys concentrates on the code optimizations aspects in this transformation chain, restricting to compilation (transforming a program to a program) for embedded processors and to high-level synthesis (transforming a program into a circuit description) for FPGAs.

Actually, it is not a surprise to see compilation and high-level synthesis getting closer. Now that high-level synthesis has grown up sufficiently to be able to rely on placing & routing tools, or even to synthesize C-like languages, standard techniques for back-end code generation (register allocation, instruction selection, instruction scheduling, software pipelining) are used in HLS tools. At the higher-level, programming languages for programmable parallel platforms share many aspects with high-level specification languages for HLS, for example, the description and manipulations of nested loops, or the model of computation/communication (e.g., Kahn process networks). In all aspects, the frontier between software and hardware is vanishing. For example, in terms of architecture, customized processors (with processor extension as proposed by Tensilica) share features with both general-purpose processors and hardware accelerators. FPGAs are both hardware and software as they are fed with "programs" representing their hardware configurations. In other words, this convergence in

code optimizations explains why Compsys studies both program compilation and high-level synthesis. Beside, Compsys has a tradition of building free software tools for linear programming and optimization in general, and will continue it, as needed for our current research.

3.2. Back-end code optimizations for embedded processors

Participants: Benoit Boissinot, Florian Brandner, Quentin Colombet, Alain Darte, Fabrice Rastello.

Compilation is an old activity, in particular back-end code optimizations. We first give some elements that explain why the development of embedded systems makes compilation come back as a research topic. We then detail the code optimizations that we are interested in, both for aggressive and just-in-time compilation.

3.2.1. Embedded systems and the revival of compilation & code optimizations

Applications for embedded computing systems generate complex programs and need more and more processing power. This evolution is driven, among others, by the increasing impact of digital television, the first instances of UMTS networks, and the increasing size of digital supports, like recordable DVD, and even Internet applications. Furthermore, standards are evolving very rapidly (see for instance the successive versions of MPEG). As a consequence, the industry has rediscovered the interest of programmable structures, whose flexibility more than compensates for their larger size and power consumption. The appliance provider has a choice between hard-wired structures (Asic), special-purpose processors (Asip), or (quasi) general-purpose processors (DSP for multimedia applications). Our cooperation with STMicroelectronics leads us to investigate the last solution, as implemented in the ST100 (DSP processor) and the ST200 (VLIW DSP processor) family for example. Compilation and, in particular, back-end code optimizations find a second life in the context of such embedded computing systems.

At the heart of this progress is the concept of *virtualization*, which is the key for more portability, more simplicity, more reliability, and of course more security. This concept, implemented through binary translation, just-in-time compilation, etc., consists in hiding the architecture-dependent features as far as possible during the compilation process. It has been used for quite a long time for servers such as HotSpot, a bit more recently for workstations, and it is quite recent for embedded computing for reasons we now explain.

As previously mentioned, the definition of "embedded systems" is rather imprecise. However, one can at least agree on the following features:

- even for processors that are programmable (as opposed to hardware accelerators), processors have some architectural specificities, and are very diverse;
- many processors (but not all of them) have some limited resources, in particular in terms of memory;
- for some processors, power consumption is an issue;
- in some cases, aggressive compilation (through cross-compilation) is possible, and even highly desirable for important functions.

This diversity is one of the reason why virtualization, which starts to be more mature, is becoming more and more common in programmable embedded systems, in particular through CIL (a standardization of MSIL). This implies a late compilation of programs, through just-in-time (JIT), including dynamic compilation. Some people even think that dynamic compilation, which can have more information because performed at run-time, can outperform the performances of "ahead-of-time" compilation.

Performing code generation (and some higher-level optimizations) in a late phase is potentially advantageous, as it can exploit architectural specificities and run-time program information such as constants and aliasing, but it is more constrained in terms of time and available resources. Indeed, the processor that performs the late compilation phase is, *a priori*, less powerful (in terms of memory for example) than a processor used for cross-compilation. The challenge is thus to spread the compilation process in time by deferring some optimizations ("deferred compilation") and by propagating some information for those whose computation is expensive ("split compilation"). Classically, a compiler has to deal with different intermediate representations (IR) where high-level information (i.e., more target-independent) co-exist with low-level information. The split

compilation has to solve a similar problem where, this time, the compactness of the information representation, and thus its pertinence, is also an important criterion. Indeed, the IR is evolving not only from a target-independent description to a target-dependent one, but also from a situation where the compilation time is almost unlimited (cross-compilation) to one where any type of resource is limited. This is also a reason why static single assignment (SSA) is becoming specific to embedded compilation, even if it was first used for workstations. Indeed, SSA is a sparse (i.e., compact) representation of liveness information. In other words, if time constraints are common to all JIT compilers (not only for embedded computing), the benefit of using SSA is also in terms of its good ratio pertinence/storage of information. It also enables to simplify algorithms, which is also important for increasing the reliability of the compiler.

In addition, this continuum of compilation strategies should integrate the need for exploiting the parallel computing resources that all recent (and future) architectures provide. A solution is to develop domain-specific languages (DSL), which adds yet another dimension to the problem of designing intermediate representation.

We now give more details on the code optimizations we want to consider and on the methodology we want to follow.

3.2.2. Aggressive and just-in-time optimizations of assembly-level code

Compilation for embedded processors is difficult because the architecture and the operations are specially tailored to the task at hand, and because the amount of resources is strictly limited. For instance, the potential for instruction level parallelism (SIMD, MMX), the limited number of registers and the small size of the memory, the use of direct-mapped instruction caches, of predication, but also the special form of applications [22] generate many open problems. Our goal is to contribute to their understanding and their solutions.

As previously explained, compilation for embedded processors include both aggressive and just in time (JIT) optimizations. Aggressive compilation consists in allowing more time to implement costly solutions (so, looking for complete, even expensive, studies is mandatory): the compiled program is loaded in permanent memory (ROM, flash, etc.) and its compilation time is not significant; also, for embedded systems, code size and energy consumption usually have a critical impact on the cost and the quality of the final product. Hence, the application is cross-compiled, in other words, compiled on a powerful platform distinct from the target processor. Just-in-time compilation corresponds to compiling applets on demand on the target processor. For compatibility and compactness, the source languages are CIL or Java. The code can be uploaded or sold separately on a flash memory. Compilation is performed at load time and even dynamically during execution. Used heuristics, constrained by time and limited resources, are far from being aggressive. They must be fast but smart enough.

Our aim is, in particular, to find exact or heuristic solutions to *combinatorial* problems that arise in compilation for VLIW and DSP processors, and to integrate these methods into industrial compilers for DSP processors (mainly ST100, ST200, Strong ARM). Such combinatorial problems can be found for example in register allocation, in opcode selection, or in code placement for optimization of the instruction cache. Another example is the problem of removing the multiplexer functions (known as ϕ functions) that are inserted when converting into SSA form. These optimizations are usually done in the last phases of the compiler, using an assembly-level intermediate representation. In industrial compilers, they are handled in independent phases using heuristics, in order to limit the compilation time. We want to develop a more global understanding of these optimization problems to derive both aggressive heuristics and JIT techniques, the main tool being the SSA representation.

In particular, we want to investigate the interaction of register allocation, coalescing, and spilling, with the different code representations, such as SSA. One of the challenging features of today's processors is predication [27], which interferes with all optimization phases, as the SSA form does. Many classical algorithms become inefficient for predicated code. This is especially surprising, since, besides giving a better trade-off between the number of conditional branches and the length of the critical path, converting control dependences into data dependences increases the size of basic blocks and hence creates new opportunities for local optimization algorithms. One has first to adapt classical algorithms to predicated code [28], but also to study the impact of predicated code on the whole compilation process.

As mentioned in Section 2.3, a lot of progress has already been done in this direction in our past collaborations with STMicroelectronics. In particular, the goal of the Sceptre project was to revisit, in the light of SSA, some code optimizations in an aggressive context, i.e., by looking for the best performances without limiting, *a priori*, the compilation time and the memory usage. One of the major results of this collaboration was to show that it is possible to exploit SSA to design a register allocator in two phases, with one spilling phase relatively target-independent, then the allocator itself, which takes into account architectural constraints and optimizes other aspects (in particular, coalescing). This new way of considering register allocation has shown its interest for aggressive static compilation. But it offers three other perspectives:

- A simplification of the allocator, which again goes toward a more reliable compiler design, based on static single assignment.
- The possibility to handle the hardest part, the spilling phase, as a preliminary phase, thus a good candidate for split compilation.
- The possibility of a fast allocator, with a much higher quality than usual JIT approaches such as "linear scan", thus suitable for virtualization and JIT compilation.

These additional possibilities have not been fully studied or developed yet. The objective of our new contract with STMicroelectronics, called Mediacom, is to address them. More generally, we want to continue to develop our activity on code optimizations, exploiting SSA properties, following our two-phases strategy:

- First, revisit code optimizations in an aggressive context to develop better strategies, without eliminating too quickly solutions that may have been considered as too expensive in the past.
- Then, exploit the new concepts introduced in the aggressive context to design better algorithms in a JIT context, focusing on the speed of algorithms and their memory footprint, without compromising too much on the quality of the generated code.

We want to consider more code optimizations and more architectural features, such as registers with aliasing, predication, and, possibly in a longer term, vectorization/parallelization again.

3.3. Program analysis and transformations for high-level synthesis

Participants: Christophe Alias, Alain Darte, Paul Feautrier, Laure Gonnord, Alexandru Plesco.

3.3.1. High-Level Synthesis Context

High-level synthesis has become a necessity, mainly because the exponential increase in the number of gates per chip far outstrips the productivity of human designers. Besides, applications that need hardware accelerators usually belong to domains, like telecommunications and game platforms, where fast turn-around and time-to-market minimization are paramount. We believe that our expertise in compilation and automatic parallelization can contribute to the development of the needed tools.

Today, synthesis tools for FPGAs or ASICs come in many shapes. At the lowest level, there are proprietary Boolean, layout, and place and route tools, whose input is a VHDL or Verilog specification at the structural or register-transfer level (RTL). Direct use of these tools is difficult, for several reasons:

- A structural description is completely different from an usual algorithmic language description, as it is written in term of interconnected basic operators. One may say that it has a spatial orientation, in place of the familiar temporal orientation of algorithmic languages.
- The basic operators are extracted from a library, which poses problems of selection, similar to the instruction selection problem in ordinary compilation.
- Since there is no accepted standard for VHDL synthesis, each tool has its own idiosyncrasies, and report its results in a different format. This makes it difficult to build portable HLS tools.
- HLS tools have trouble handling loops. This is particularly true for logic synthesis systems, where loops are systematically unrolled (or considered as sequential) before synthesis. An efficient treatment of loops needs the polyhedral model. This is where past results from the automatic parallelization community are useful.

- More generally, a VHDL specification is too low level to allow the designer to perform, easily, higher-level code optimizations, especially on multi-dimensional loops and arrays, which are of paramount importance to exploit parallelism, pipelining, and perform memory optimizations.

Some intermediate tools exist that generate VHDL from a specification in restricted C, both in academia (such as SPARK, Gaut, UGH, CloogVHDL, and in industry (such as C2H), CatapultC, PICO Express. All these tools use only the most elementary form of parallelization, equivalent to instruction-level parallelism in ordinary compilers, with some limited form of block pipelining. Targeting one of these tools for low-level code generation, while we concentrate on exploiting loop parallelism, might be a more fruitful approach than directly generating VHDL. However, it may be that the restrictions they impose preclude efficient use of the underlying hardware.

Our first experiments with these HLS tools reveal two important issues. First, they are, of course, limited to certain types of input programs so as to make their design flows successful. It is a painful and tricky task for the user to transform the program so that it fits these constraints and to tune it to get good results. Automatic or semi-automatic program transformations can help the user achieve this task. Second, users, even expert users, have only a very limited understanding of what back-end compilers do and why they do not lead to the expected results. An effort must be done to analyze the different design flows of HLS tools, to explain what to expect from them, and how to use them to get a good quality of results. Our first goal is thus to develop high-level techniques that, used in front of existing HLS tools, improve their utilization. This should also give us directions on how to modify them.

More generally, we want to consider HLS as a more global parallelization process. So far, no HLS tools is capable of generating designs with communicating *parallel* accelerators, even if, in theory, at least for the scheduling part, a tool such as PICO Express could have such capabilities. The reason is that it is, for example, very hard to automatically design parallel memories and to decide the distribution of array elements in memory banks to get the desired performances with parallel accesses. Also, how to express communicating processes at the language level? How to express constraints, pipeline behavior, communication media, etc.? To better exploit parallelism, a first solution is to extend the source language with parallel constructs, as in all derivations of the Kahn process networks model, including communicating regular processes (CRP, see later). The other solution is a form of automatic parallelization. However, classical methods, which are mostly based on scheduling, are not directly applicable, firstly because they pay poor attention to locality, which is of paramount importance in hardware. Beside, their aim is to extract all the parallelism in the source code; they rely on the runtime system to tailor the parallelism degree to the available resources. Obviously, there is no runtime system in hardware. The real challenge is thus to invent new scheduling algorithms that take both resource and locality into account, and then to infer the necessary hardware from the schedule. This is probably possible only for programs that fit into the polytope model.

In summary, as for our activity on back-end code optimizations, which is decomposed into two complementary activities, aggressive and just-in-time compilation, we focus our activity on high-level synthesis on two aspects:

- Developing high-level transformations, especially for loops and memory/communication optimizations, that can be used in front of HLS tools so as to improve their use.
- Developing concepts and techniques in a more global view of high-level synthesis, starting from specification languages down to hardware implementation.

We now give more details on the program optimizations and transformations we want to consider and on our methodology.

3.3.2. Specifications, Transformations, Code Generation for High-Level Synthesis

Before contributing to high-level synthesis, one has to decide which execution model is targeted and where to intervene in the design flow. Then one has to solve scheduling, placement, and memory management problems. These three aspects should be handled as a whole, but present state of the art dictates that they be treated

separately. One of our aims will be to find more comprehensive solutions. The last task is code generation, both for the processing elements and the interfaces between FPGAs and the host processor.

There are basically two execution models for embedded systems: one is the classical accelerator model, in which data is deposited in the memory of the accelerator, which then does its job, and returns the results. In the streaming model, computations are done on the fly, as data flow from an input channel to the output. Here, data is never stored in (addressable) memory. Other models are special cases, or sometime compositions of the basic models. For instance, a systolic array follows the streaming model, and sometime extends it to higher dimensions. Software radio modems follow the streaming model in the large, and the accelerator model in detail. The use of first-in first-out queues (FIFO) in hardware design is an application of the streaming model. Experience shows that designs based on the streaming model are more efficient that those based on memory. One of the point to be investigated is whether it is general enough to handle arbitrary (regular) programs. The answer is probably negative. One possible implementation of the streaming model is as a network of communicating processes either as Kahn process networks (FIFO based) or as our more recent model of communicating regular processes (CRP, memory based). It is an interesting fact that several researchers have investigated translation from process networks [23] and to process networks [29], [30].

Kahn process networks (KPN) were introduced 30 years ago as a notation for representing parallel programs. Such a network is built from processes that communicate via perfect FIFO channels. Because the channel histories are deterministic, one can define a semantics and talk meaningfully about the equivalence of two implementations. As a bonus, the dataflow diagrams used by signal processing specialists can be translated on-the-fly into process networks. The problem with KPNs is that they rely on an asynchronous execution model, while VLIW processors and FPGAs are synchronous or partially synchronous. Thus, there is a need for a tool for synchronizing KPNs. This is best done by computing a schedule that has to satisfy data dependences within each process, a causality condition for each channel (a message cannot be received before it is sent), and real-time constraints. However, there is a difficulty in writing the channel constraints because one has to count messages in order to establish the send/receive correspondence and, in multi-dimensional loop nests, the counting functions may not be affine. In order to bypass this difficulty, one can define another model, *communicating regular processes* (CRP), in which channels are represented as write-once/read-many arrays. One can then dispense with counting functions. One can prove that the determinacy property still holds. As an added benefit, a communication system in which the receive operation is not destructive is closer to the expectations of system designers.

The main difficulty with this approach is that ordinary programs are usually not constructed as process networks. One needs automatic or semi-automatic tools for converting sequential programs into process networks. One possibility is to start from array dataflow analysis [24]. Each statement (or group of statements) may be considered a process, and the source computation indicates where to implement communication channels. Another approach attempts to construct threads, i.e. pieces of sequential code with the smallest possible interactions. In favorable cases, one may even find outermost parallelism, i.e. threads with no interactions whatsoever. Here, communications are associated to so-called uncut dependences, i.e. dependences which cross thread boundaries. In both approaches, the main question is whether the communications can be implemented as FIFOs, or need a reordering memory. One of our research directions will be to try to take advantage of the reordering allowed by dependences to force a FIFO implementation.

Whatever the chosen solution (FIFO or addressable memory) for communicating between two accelerators or between the host processor and an accelerator, the problems of optimizing communication between processes and of optimizing buffers have to be addressed. Many local memory optimization problems have already been solved theoretically. Some examples are loop fusion and loop alignment for array contraction and for minimizing the length of the reuse vector [26], techniques for data allocation in scratch-pad memory, or techniques for folding multi-dimensional arrays [21]. Nevertheless, the problem is still largely open. Some questions are: how to schedule a loop sequence (or even a process network) for minimal scratch-pad memory size? How is the problem modified when one introduces unlimited and/or bounded parallelism? How does one take into account latency or throughput constraints, or bandwidth constraints for input and output channels? All loop transformations are useful in this context, in particular loop tiling, and may be applied

either as source-to-source transformations (when used in front of HLS tools) or as transformations to generate directly VHDL codes. One should keep in mind that theory will not be sufficient to solve these problems. Experiments are required to check the relevance of the various models (computation model, memory model, power consumption model) and to select the most important factors according to the architecture. Besides, optimizations do interact: for instance reducing memory size and increasing parallelism are often antagonistic. Experiments will be needed to find a global compromise between local optimizations.

Finally, there remains the problem of code generation for accelerators. It is a well-known fact that modern methods for program optimization and parallelization do not generate a new program, but just deliver blueprints for program generation, in the form, e.g., of schedules, placement functions, or new array subscripting functions. A separate code generation phase must be crafted with care, as a too naïve implementation may destroy the benefits of high-level optimization. There are two possibilities here as suggested before; one may target another high-level synthesis tool, or one may target directly VHDL. Each approach has its advantages and drawbacks. However, in both situations, all such tools, including VHDL but not only, require that the input program respects some strong constraints on the code shape, array accesses, memory accesses, communication protocols, etc. Furthermore, to get the tool do what the user wants requires a lot of program tuning, i.e., of program rewriting. What can be automated in this rewriting process? Semi-automated? Our partnership with STMicroelectronics (synthesis) should help us answer such a question, considering both industrial applications and industrial HLS tools. Also, in the hope of extending the tools Gaut and Ugh beyond this stage, an informal working group (Coach), with members from Lab-STICC (Lorient), Asim (UPMC), Cairn (IRISA), Tima (IMAG), and Compsys has started meeting regularly, with the goal of submitting a revised ANR proposal at the next opportunity.

4. Application Domains

4.1. Application Domains

Keywords: embedded computing systems, compilation, high-level synthesis, compilation, program optimizations.

The previous sections describe our main activities in terms of research directions, but also places Compsys within the embedded computing systems domain, especially in Europe. We will therefore not come back here to the importance, for industry, of compilation and embedded computing systems design.

In terms of application domain, the embedded computing systems we consider are mostly used for multimedia: phones, TV sets, game platforms, etc. But, more than the final applications developed as programs, our main application is the computer itself: how the system is organized (architecture) and designed, how it is programmed (software), how programs are mapped to it (compilation and high-level synthesis).

The industry that can be impacted by our research is thus all the companies that develop embedded systems and processors, and those (the same plus other) than need software tools to map applications to these platforms, i.e., that need to use or even develop programming languages, program optimization techniques, compilers, operating systems. Compsys do not focus on all these critical parts, but our activities are connected to them.

5. Software

5.1. Introduction

This section lists and briefly describes the software developments conducted within Compsys. Most are tools that we extend and maintain over the years. They now concern two activities only: a) the development of tools linked to polyhedra and loop/array transformations, b) the development of algorithms within the back-end compiler of STMicroelectronics.

The previous annual reports contain descriptions of Pip, a tool for parametric integer linear programming, of the Polylib tool, a C library of polyhedral operations, and of MMAlpha a circuit synthesis tool for systolic arrays. These tools, developed by members or past members of Compsys, are not maintained anymore in Compsys, but are extended by other groups. They have been important tools in the development of the "polytope model", which is now widely accepted: it is used by Inria projects-teams Cairn and Alchemy, PIPS at École des Mines de Paris, Suif from Stanford University, Compaan at Berkeley and Leiden, PiCo from the HP Labs (continued as PicoExpress by Synfora), the DTSE methodology at Imec, Sadayappan's group at Ohio State University, Rajopadhye's group at Colorado State's University, etc. These groups are research projects, but the increased involvement of industry (Hewlett Packard, Philips, Synfora, Reservoir Labs) is a favorable factor. Polyhedra are also used in test and certification projects (Verimag, Lande, Vertecs). More recently, several compiler groups have shown their interest in polyhedral methods: the GCC group and Reservoir Labs in the USA, which develops a compiler fully-based on the polytope model and on the techniques we introduced for loop and array transformations. Now that these techniques are well-established and disseminated by other groups (in particular Alchemy), we prefer to focus on the development of new techniques and tools, which are described here.

The other activity concerns the developments within the compiler of STMicroelectronics. These are not stand-alone tools, which could be used externally, but algorithms and data structures implemented inside the LAO back-end compiler, year after year, with the help of STMicroelectronics colleagues. As these are also important developments, it is worth mentioning them in this section. They are also completed by important efforts for integration and evaluation within the complete STMicroelectronics toolchain. They concern exact methods (ILP-based), algorithms for aggressive optimizations, techniques for just-in-time compilation, and for improving the design of the compiler.

5.2. Pip

Participants: Cédric Bastoul [MCF, IUT d'Orsay], Paul Feautrier.

Paul Feautrier is the main developer of Pip (Parametric Integer Programming) since its inception in 1988. Basically, Pip is an "all integer" implementation of the Simplex, augmented for solving integer programming problems (the Gomory cuts method), which also accepts parameters in the non-homogeneous term. Pip is freely available under the GPL at http://www.piplib.org. Pip is widely used in the automatic parallelization community for testing dependences, scheduling, several kind of optimizations, code generation, and others. Beside being used in several parallelizing compilers, Pip has found applications in some unconnected domains, as for instance in the search for optimal polynomial approximations of elementary functions (see the Inria project Arénaire).

5.3. Syntol

Participants: Hadda Cherroun [Former PhD student in Compsys], Paul Feautrier.

Syntol is a modular process network scheduler. The source language is C augmented with specific constructs for representing communicating regular process (CRP) systems. The present version features a syntax analyzer, a semantic analyzer to identify DO loops in C code, a dependence computer, a modular scheduler, and interfaces for CLooG (loop generator developed by C. Bastoul) and Cl@k (see Sections 5.4 and 5.5). The dependence computer now handles casts, records (`structures`), and the modulo operator in subscripts and conditional expressions. The latest developments are, firstly, a new code generator, and secondly, several experimental tools for the construction of bounded parallelism programs.

- The new code generator, based on the ideas of Boulet and Feautrier [20], generates a counter automaton that can be presented as a C program, as a rudimentary VHDL program at the RTL level, as an automaton in the Aspic input format, or as a drawing specification for the DOT tool.
- Hardware synthesis can only be applied to bounded parallelism programs. Our present aim is to construct threads with the objective of minimizing communications and simplifying synchronization. The distribution of operations among threads is specified using a placement function, which is found using techniques of linear algebra and combinatorial optimization.

5.4. Cl@k

Participants: Christophe Alias, Fabrice Baray [Mentor, Former post-doc in Compsys], Alain Darte.

A few years ago, we identified new mathematical tools useful for the automatic derivation of array mappings that enable memory reuse, in particular the notions of admissible lattice and of modular allocation (linear mapping plus modulo operations). Fabrice Baray, former post-doc Inria under Alain Darte's supervision, developed a stand-alone optimization software tool in 2005-2006, called Cl@k (for Critical LAttice Kernel), that computes or approximates the critical lattice for a given 0-symmetric polytope. (An admissible lattice is a lattice whose intersection with the polytope is reduced to 0; a critical lattice is an admissible lattice with minimal determinant.) The array contraction technique itself has been implemented by Christophe Alias in a tool called Bee. Bee uses Rose, as a parser, analyzes the lifetime of elements of the arrays to be compressed, and builds the necessary input for Cl@k, i.e., the 0-symmetric polytope of conflicting differences. Then, Bee computes the array contraction mapping from the lattice provided by Cl@k and generates the final program with contracted arrays. See previous reports for more details on the underlying theory. Cl@k can be viewed as a complement to the Polylib suite, enabling yet another kind of optimizations on polyhedra. Bee is the complement of Cl@k in terms of its application to memory reuse. Bee is now a stand-alone tool that contain more and more features for program analysis and loop transformations.

5.5. Bee

Participants: Christophe Alias, Alain Darte.

Historically, BEE was a source-to-source translator performing array contraction. Meanwhile, many program analyses and features have been added to BEE, which is now a *source-to-source compiler*. BEE provides many facilities to quickly prototype polyhedral program analyses as described hereafter.

- **C front-end.** Based on EDG (*via* Rose), an industrial C/C++ parser from Edison group used in Intel compilers.
- **XML front-end.** Optional feature to avoid license problems with Rose and EDG. This feature has been used to connect the array contraction module of BEE to GECOS, a source-to-source compiler developed in the CAIRN Inria team.
- **Polyhedral intermediate representation.** Automatic extraction of polyhedral domains and affine access functions.
- **Symbolic layer** on the libraries POLYLIB (set operations on polyhedra) and PIPLIB (parameterized ILP). These two last features simplify drastically the developer task.
- **C back-end.** With both the method of Quilleré and Rajopadhye (*via* Cloog http://www.cloog.org), and the method of Boulet and Feautrier [20].
- **VHDL back-end.** With the Boulet-Feautrier method.
- **Instance-wise dataflow analysis.** Data dependence analysis, array dataflow analysis (ADA), array lifetime analysis (ALA), array region analysis. The ADA of BEE has been used, improved, and extended to form the stand-alone library FADALIB (http://www.prism.uvsq.fr/~bem/fadalib/).
- **Instance-wise program analysis.** Program termination, symbolic complexity estimation, array contraction.

BEE is – to our knowledge – the only source-to-source compiler with a complete array contraction method. BEE rewrites a kernel written in C to reduce the size of arrays, bridging the gap between the theoretical framework described in [21] and implemented in Cl@k, and effective program transformations for array contraction. For that, a precise lifetime analysis for arrays has been designed and implemented. After being determined by Cl@k, the allocations are then translated back from the critical integer lattices into real code: the arrays are remapped thanks to a linear (modular) allocation function ($a[\overrightarrow{i}] \mapsto a'[A\overrightarrow{i} \mod \overrightarrow{b}]$) that collapses array cells that do not live at the same time.

BEE also provides a language of pragmas to specify the kernel to be analyzed, the arrays to be contracted, and (optionally) the affine schedule of the kernel. The latter feature enlarges the application field of array contraction to parallel programs. For instance, it is possible to mark a loop to be software-pipelined (with an affine schedule), and to let BEE find an optimized array contraction. But the most important application is the ability to optimize communicating regular processes (CRP). Given a schedule for every process, BEE can compute an optimal size for the channels, together with their access functions (the corresponding allocations). We currently use this feature in source-to-source transformations for high-level synthesis (see Section 3.3).

BEE has been fully implemented by Christophe Alias and represents more than 7000 lines of code. Christophe Alias is responsible for the maintenance of BEE.

5.6. Chuba

Participants: Christophe Alias, Alain Darte, Alexandru Plesco.

Chuba is a source-to-source tool implementing the algorithms described in the PhD thesis of Alexandru Plesco. Chuba is a source-level optimizer for the high-level synthesis compiler C2H. It takes as input the source C code of the kernel to optimize and generates a C2H-compliant C code describing a system of multiple communicating accelerators. The data transfers from the external DDR memory are optimized: they are performed by blocks of computations, obtained thanks to tiling techniques, and, in each block, data are fetched by block to reduce the penalty due to line changes in the DDR accesses. Four accelerators achieve data transfers (two from the DDR, two to the DDR) in a macro-pipeline fashion, so that data transfers and computations (performed by a fifth accelerator) are overlapped. It is interesting to mention that the program analysis and optimizations implemented in Chuba address a problem that is also very relevant in the context of GPGPUs.

So far, the backend of Chuba can only generate code for C2H but we intend to adapt it so that it can be used by other HLS tools such as CatapultC. The backend exploits the semantics of the C2H synthesis of C and generates multiple functions for each accelerator. The functions contain multiple software-pipelined loops, iterating over the tiles and inside the tiles. To not pay, at each iteration of the outer loops, the latency of the inner software pipelined loops, all loops iterating over tiles and all loop iterating inside tiles are linearized using the Boulet-Feautrier method [20].

Chuba has been fully implemented by Christophe Alias, using the compiler infrastructure BEE. It represents more than 1800 lines of C++. The reduced size of Chuba is mainly due to the high-level abstractions provided by BEE.

5.7. RanK

Participants: Christophe Alias, Alain Darte, Paul Feautrier, Laure Gonnord.

RanK is a software tool that can prove the termination of a program (in some cases) by computing a *ranking function*, i.e., a mapping from the operations of the program to a well-founded set that *decreases* as the computation advances. In case of success, RanK can also provide an upper bound of the worst-case time complexity of the program as a symbolic affine expression involving the input variables of the program (parameters), when it exists. In case of failure, RanK tries to prove the non-termination of the program and then to exhibit a counter-example input. This last feature is of great help for program understanding and debugging, and has already been experimented.

The input of RanK is an integer automaton, computed by `C2fsm` (see hereafter), representing the control structure of the program to check. RanK uses the Aspic tool, developed by Laure Gonnord during her PhD thesis, to compute automaton invariants. RanK has been used to discover successfully the worst-case time complexity of many benchmarks programs of the community. It uses the libraries Piplib and Polylib.

RanK has been fully implemented by Christophe Alias, using the compiler infrastructure BEE and represents more than 3000 lines of C++. RanK uses several high-level symbolic features developped in BEE.

5.8. C2fsm

Participant: Paul Feautrier.

`C2fsm` is a general tool that converts an arbitrary C program into a counter automaton. This tool reuses the parser and pre-processor of Syntol, which has been greatly extended to handle `while` and `do while` loops, `goto`, `break`, and `continue` statements. `C2fsm` reuses also part of the code generator of Syntol and has several output formats, including FAST (the input format of Aspic), a rudimentary VHDL generator, and a DOT generator which draws the output automaton. `C2fsm` is also able to do elementary transformations on the automaton, such as eliminating useless states, transitions and variables, simplifying guards, or selecting cut-points, i.e., program points on loops that can be used by RanK to prove program termination.

5.9. LAO developments in aggressive compilation

Participants: Benoit Boissinot, Florent Bouchez, Florian Brandner, Quentin Colombet, Alain Darte, Benoit Dupont-de-Dinechin [Kalray], Christophe Guillon [STMicroelectronics], Sebastian Hack [Former post-doc in Compsys], Fabrice Rastello, Cédric Vincent [Former student in Compsys].

Our aggressive optimization techniques are all implemented in stand-alone experimental tools (as for example for register coalescing algorithms) or within LAO, the back-end compiler of STMicroelectronics, or both. They concern SSA construction and destruction, instruction-cache optimizations, register allocation. Here, we report only our more recent activities, which concern register allocation.

Our developments on register allocation with the STMicroelectronics compiler started when Cédric Vincent (bachelor degree, under Alain Darte supervision) developed a complete register allocator in LAO, the assembly-code optimizer of STMicroelectronics. This was the first time a complete implementation was done with success, outside the MCDT (now CEC) team, in their optimizer. Since then, new developments are constantly done, in particular by Florent Bouchez, advised by Alain Darte and Fabrice Rastello, as part of his master internship and PhD thesis. In 2009, Quentin Colombet started to develop and integrate into the main trunk of LAO a full implementation of a two-phases register allocation. This implementation now includes two different decoupled spilling phases, the first one as described in Sebastian Hack's PhD thesis and a new ILP-based solution (see Section 6.5). It also includes an up-to-date graph-based register coalescing. Finally, since all these optimizations take place under SSA form, it includes also a mechanism for going out of colored-SSA (register-allocated SSA) form that can handle critical edges and does further optimizations (see Section 6.3).

5.10. LAO developments in JIT compilation

Participants: Benoit Boissinot, Florian Brandner, Alain Darte, Benoit Dupont-de-Dinechin [Kalray], Christophe Guillon [STMicroelectronics], Fabrice Rastello.

The other side of our work in the STMicroelectronics compiler LAO has been to adapt the compiler to make it more suitable for JIT compilation. This means lowering the time and space complexity of several algorithms. In particular we implemented our translation out-of-SSA method, and we programmed and tested various ways to compute the liveness information as described in Section 6.6. Recent efforts (see Section 6.8) also focused on developing a tree-scan register allocator for the JIT part of the compiler, in particular a JIT conservative coalescing. The technique is to bias the tree-scan coalescing, taking into account register constraints, with the result of a JIT aggressive coalescing.

6. New Results

6.1. Introduction

This section presents the results obtained by Compsys in 2010. For clarity, some earlier work is also recalled, when results were continued or extended in 2010.

6.2. Split register allocation: linear complexity without performance penalty

Participants: Albert Cohen [Inria, Alchemy], Boubacar Diouf [Université Paris Sud, Alchemy], Fabrice Rastello.

Just-in-time compilers are catching up with ahead-of-time frameworks, stirring the design of more efficient algorithms and more elaborate intermediate representations. They rely on continuous, feedback-directed (re-)compilation frameworks to adaptively select a limited set of hot functions for aggressive optimization. Leaving the hottest functions aside, (quasi-)linear complexity remains the driving force structuring the design of just-in-time optimizers.

We addressed the (spill-everywhere) register allocation problem, showing that linear complexity does not imply lower code quality. We presented a split compiler design, where a more expensive ahead-of-time analysis guides lightweight just-in-time optimizations. A split register allocator can be very aggressive in its offline stage (even optimal), producing a semantically equivalent digest through bytecode annotations that can be processed by a lightweight online stage. The algorithmic challenges are threefold: (sub-)linear-size annotation, linear-time online stage, minimal loss of code quality. In most cases, portability of the annotation is an important fourth challenge.

We proposed a split register allocator meeting these four challenges, where a compact annotation derived from an optimal integer linear program drives a linear-time algorithm near optimality. We studied the robustness of this algorithm to variations in the number of physical registers and to variations in the target instruction set. Our method has been implemented in JikesRVM and evaluated on standard benchmarks. The split register allocator achieves wall-clock improvements reaching 4.2% over the baseline allocator, with annotations spanning a fraction of the bytecode size.

This work is part of a collaboration with the Alchemy Inria project-team. It has been presented at the HiPEAC'10 conference [10].

6.3. Parallel copy motion and critical edge splitting

Participants: Florent Bouchez, Quentin Colombet, Alain Darte, Christophe Guillon [STMicroelectronics], Fabrice Rastello.

Recent results on the SSA form led to the design of heuristics based on tree scans with two decoupled phases, one for spilling, one for splitting/coloring/coalescing. Another class of register allocators, well-suited for JIT compilation, are those based on linear scans. Most of them perform coalescing poorly but also do live-range splitting (mostly on control-flow edges) to avoid spilling. This leads to a large amount of register-to-register copies inside basic blocks but also, implicitly, on critical edges, i.e., edges that flow from a block with several successors to a block with several predecessors.

We proposed a new back-end optimization that we call parallel copy motion. The technique is to move copy instructions in a register-allocated code from a program point, possibly an edge, to another. In contrast with a classical scheduler that must preserve data dependences, our copy motion also permutes register assignments so that a copy can "traverse" all instructions of a basic block, except those with conflicting register constraints. Thus, parallel copies can be placed either where the scheduling has some empty slots (for multiple-issues architectures), or where fewer copies are necessary because some variables are dead at this point. Moreover, to the cost of some code compensations (namely, the reverse of the copy), a copy can also be moved out from a critical edge. This provides a simple solution to avoid critical-edge splitting, especially useful when the compiler cannot split it, as it is the case for the so-called abnormal edges. This compensation technique also enables the scheduling/motion of the copy in the successor or predecessor basic block.

Experiments with the SPECint benchmarks suite and our own benchmark suite show that we can now apply broadly an SSA-based register allocator: all procedures, even with abnormal edges, can be treated. Simple strategies for moving copies from edges and locally inside basic blocks show significant average improvements (4% for SPECint and 3% for our suite), with no degradation. It let us believe that the approach is promising, and not only for improving coalescing in fast register allocators. This work has been presented at the SCOPES'10 conference [7].

6.4. Static single information form: debunking

Participants: Benoit Boissinot, Philip Brisk [EPFL, Lausanne], Alain Darte, Fabrice Rastello.

The static single information (SSI) form, proposed by Ananian, then in a more general form by Singer, is an extension of the static single assignment (SSA) form. The fact that interference graphs for procedures represented in SSA form are chordal is a nice property that initiated our work on register allocation under SSA form. Several interesting results have also been shown for SSI concerning liveness analysis and representation of live-ranges of variables, which could make SSI appealing for just-in-time compilation. In particular, the chordal property for SSA has motivated Brisk and Sarrafzadeh is 2007 to prove a similar result concerning SSI: the interference graph is an interval graph. Unfortunately, previous literature on SSI is sparse, appears to be partly incorrect, including the proof on interval graphs, and based on several inaccurate claims and theories (in particular the program structure tree of Johnson et al.).

We corrected some of the mistakes that have been made on SSI. Our main result is a complete proof that, even for the most general definition of SSI, it is possible to define a total order on basic blocks, and thus on program points, so that live-ranges of variables correspond to intervals. The proof is based on the notion of loop nesting forest, as formulated by Ramalingam. This study, which is the result of an informal collaboration with Philip Brisk from the University of Lausanne (EPFL), has led to a journal publication in ACM TECS [3]. It is also one of the main results of the PhD thesis of Benoit Boissinot [1].

6.5. "Optimal" formulation of register spilling

Participants: Florian Brandner, Quentin Colombet, Alain Darte, Fabrice Rastello.

The motivation of this work was to develop an optimal spilling algorithm, based on integer linear programming (ILP), to be used to evaluate heuristics and to better understand the traps in which they can fall.

Optimizing the placement of LOAD and STORE instructions (spill) is the key to get good performances in compute-intensive codes and to avoid memory transfers, which impact time and power consumption. Performing register allocation in two decoupled phases enables the design of faster and better spilling heuristics. We developed an ILP formulation for "optimal" spilling optimization under SSA, which models the problem in a finer way than previous approaches. In particular, we can model the fact that a given variable can reside in more than one storage location (different registers and in memory). This formulation has been fully implemented in the LAO back-end compiler. Several difficulties have been revealed, which were expected but never precisely identified: they are due to the MOVE instructions that propagate values between registers and to interactions with post optimization phases, such as some peephole optimizations and post-pass scheduling. More work has to be done to improve our formulation even further and to derive, from a study of benchmarks, some good criteria to drive spilling heuristics.

This work is still in progress.

6.6. Fast computation of liveness sets

Participants: Benoit Boissinot, Florian Brandner, Alain Darte, Benoit Dupont-de-Dinechin [Kalray], Fabrice Rastello.

We revisited the problem of computing liveness sets (live-in and live-out sets of basic blocks) for all variables of strict SSA-form programs. In strict SSA, the definition of a variable dominates all its uses, so the backward data-flow analysis for computing liveness sets can be simplified. Our first contribution was the design of a fast non-iterative data-flow algorithm, which exploits the SSA properties so that only two passes (backward, then forward) are necessary to compute liveness sets. Our solution relies on the use of a loop nesting forest (as defined by Ramalingam) and, unlike structure-based liveness algorithms, can handle any control-flow graph, even non-reducible. A second – maybe more natural – approach is to identify, one path at a time, all paths from a use of a variable to its (unique) definition. Such a strategy is used in the LLVM compiler and in Appel's "Tiger book". Our second contribution is to show how to extend and optimize these algorithms for computing liveness sets, one variable at a time using adequate data structures.

Finally, we demonstrated and compared the efficiency of our solutions using different implementations of liveness sets based on bitsets and ordered pointer sets. The algorithms were implemented in the LAO back-end of the STMicroelectronics compiler and subsequently evaluated using the SPECINT 2000 benchmark suite. The experiments show that our new approach outperforms the standard data-flow liveness analysis, as we could expect. In comparison to the variable-by-variable algorithm, we see a more differentiated picture. While processing each variable individually is clearly faster on non-optimized programs, the results for optimized programs highly depend on the liveness-set implementation. Ordered sets clearly favor the per-variable approach due to the reduced cost of set-insertion. Bitsets on the other hand perform best with the forest-based algorithm, due to improved locality and faster operations on complete sets. Overall, our new forest algorithm using bitsets performs best. Another interesting result is the fact that the variable-by-variable algorithm can easily be extended to compute liveness sets for regular (non-SSA-form) programs. The resulting algorithm is still faster than standard iterative data-flow liveness analysis.

6.7. Decoupled graph-coloring register allocation with hierarchical aliasing

Participants: Mariza Bigonha [UFMG, Brazil], Quentin Colombet, Christophe Guillon [STMicroelectronics], Fernando Pereira [UFMG, Brazil], Fabrice Rastello, Andre Tavares [UFMG, Brazil].

Decoupling spilling from register assignment, as mentioned in previous sections, has two main advantages: first, it simplifies register allocation algorithms, second, it might keep more variables in registers, instead of sending them to memory. In spite of these advantages, the decoupled model, in its basic formulation, does not handle register aliasing, a phenomenon present in architectures such as x86, ARM, and Sparc. An important obstacle is the fact that existing decoupled algorithms have to perform extensive live range splitting to deal with aliasing, increasing the input interference graphs by a quadratic factor. Such allocators would be inefficient in terms of memory consumption, compilation time, and the quality of the code they produce.

To address these issues, we introduced a number of techniques that circumvent this obstacle. We described a spill test that deals with aliasing better than Kempe's traditional simplification test. We developed heuristics to merge – or rather to avoid splitting – live ranges whenever possible, and we adapted well-known coalescing tests to the world of aliased registers. We validated our results empirically by showing how our techniques improve two well-known allocated, based on graph-coloring, that deal with aliased registers: Smith et al.'s extension of the Appel-George iterated register coalescing (IRC), and Bouchez et al.'s brute force method. Running our techniques on SPEC CPU 2000, we were able to reduce, by a factor of 4, the size of the interference graphs that the allocators would require, and we improved the quality of the IRC, in terms of proportion of copies left in the assembly program, from 1.5% to 0.54%.

6.8. Graph-coloring and tree-scan register allocation using repairing

Participants: Benoit Boissinot, Philip Brisk [University of California, United States], Quentin Colombet, Sebastian Hack [Saarland University, Germany], Fabrice Rastello.

We introduced repairing, a new technique to deal with register constraints during register allocation that avoids early (i.e., pre-pass) live-range splitting in register allocation. Early live range splitting increases the program size and thus increases the compile time significantly. Repairing ignores register constraints during allocation and repairs potential violations afterwards.

We showed how to integrate repairing into the iterated register coalescing (IRC) graph-coloring allocator without changing its architecture and implementation significantly. Furthermore, we showed how to do repairing in SSA-based decoupled tree-scan register allocators. To completely avoid the compile-time intensive coalescing phase in this case, we presented several techniques to bias the register assignment.

Our experimental evaluation focused on just-in-time (JIT) compilation where the runtime of the compiler and the code size of its data structures are of greater importance than the quality of the code that is produced. Our modification of the IRC reduces the number of vertices in the interference graph by 26% (33% for the edges) without compromising the quality of the generated code. The tree-scan algorithm is 7 times faster than the IRC while providing comparable results for the quality of the generated code.

6.9. Program analysis and communication optimizations for HLS

Participants: Christophe Alias, Alain Darte, Alexandru Plesco.

High-level synthesis (HLS) tools are now getting more mature for generating hardware accelerators with an optimized internal structure, thanks to efficient scheduling techniques, resource sharing, and finite-state machines generation. However, interfacing them with the outside world, i.e., integrating the automatically-generated hardware accelerators within the complete design, with optimized communications, so that they achieve the best throughput, remains a very hard task, reserved to expert designers. The goal of our research on HLS is to study and to develop source-to-source strategies to improve the design of these interfaces, trying to consider the HLS tool as a back-end for more advanced front-end transformations.

In a previous study, we used the Wrapit loop transformation tool (Alchemy team) on top of the Spark HLS tool to demonstrate both the importance of loop transformations as a pre-processing step to HLS tools and the difficulty to use them depending on the HLS tool features to express external communications. In 2009-2010, using the C2H HLS tool from Altera, which can synthesize hardware accelerators communicating to an external DDR-SDRAM memory, we showed that it is possible to automatically restructure the application code, to generate adequate communication processes in C, and to compile them all with C2H, so that the resulting application is highly-optimized, with full usage of the memory bandwidth.

These transformations and optimizations, which combine techniques such as double buffering, array contraction, loop tiling, software pipelining, among others, were incorporated in an automatic source-to-source transformation tool, called CHUBA, based on the polyhedral model representation. Our study shows that HLS tools can indeed be used as back-end optimizers for front-end optimizations, as it is the case for standard compilation with high-level transformations developed on top of assembly-code optimizers. We believe this is the way to go for making HLS tools viable. The first part of our study, which analyzes the C2H Altera synthesis tool and shows how communications can be optimized by hand, but in an automatizable fashion, has been published at the ASAP'10 conference [6]. The second part of our study, a complete automatization of the process, which included program analysis, program transformations, and code generation, is still unpublished, except in the PhD thesis of Alexandru Plesco [2].

6.10. Loop transformations for pipelined floating-point arithmetic operators

Participants: Christophe Alias, Bogdan Pasca [PhD student, ARENAIRE Inria Team], Alexandru Plesco.

FLOPOCO [31] is an open-source FPGA-specific generator of pipelined floating-point arithmetic operators developped in the ARENAIRE Inria team. These operators, though efficient, still need to be incorporated by hand in the final design. We studied how to compile a C program to a circuit using such pipelined arithmetic operators efficiently, that is (i) how to select properly the operators to generate, (ii) how to schedule the program to use them efficiently, and (iii) how to generate the final VHDL code for the control unit to be linked with the pipelined operators. In a way, this work can be viewed as a very fancy hardware-level instruction selection.

We proposed a preliminary solution (see [14]) for affine programs with perfectly-nested loops with uniform dependences. The pipelined operator computes the *right hand-side* expression of the unique assignment nested in the loops. Under these restrictions, it is possible to schedule the program so that the pipelined operator is kept busy: each result is available exactly at the time it is needed by the operator, avoiding the use of a temporary buffer. This is possible thanks to uniform dependences, which correspond to a constant reuse distance. Also, we proposed a method to generate the VHDL code for the control unit, according to the chosen schedule. Then, the connection with the pipelined operator is done by hand. The first experimental results on DSP kernels give promising results with a minimum of 94% efficient utilization of the pipelined operators for a complex kernel.

This is still a work in progress and many issues need to be addressed, such as how to use several operators in parallel or how to extend the program model to general nested loops with more general dependences. These extensions will require to handle properly the communications between the operators and temporary buffers. We believe that the array contraction technique developed in Compsys can be helpful in this context too.

6.11. Program termination and worst-case computational complexity

Participants: Christophe Alias, Laure Gonnord, Paul Feautrier, Alain Darte.

Our current work on program termination arose when trying to transform WHILE loops into DO loops (i.e., loops with a bounded number of iterations) so that HLS tools can accept them. Some HLS tools indeed need this information either for unrolling loops, for pipelining them, or for scheduling, at a higher-level, several pipelined designs including loops. Determining the maximal number of iterations of a WHILE loop is a particular case of determining the worst-case computational complexity (WCCC) of a function or subroutine, which is, briefly speaking, an upper-bound on the number of elementary computations that it performs. It is an abstract and architecture-independent view of the well-known worst-case execution time (WCET). Knowledge of WCET is a key information for embedded systems, as it allows the construction of a schedule and the verification that real-time constraints are met. The WCCC and WCET problems are obviously connected to the termination problem: a piece of code that does not terminate has no WCCC and no WCET.

The standard method for proving the termination of a flowchart program is to exhibit a ranking function, i.e., a function from the program states to a well-founded set, which strictly decreases at each program step. Such a function can be automatically generated by computing invariants (approximation of all possible values of program variables) for each program point and by searching for a ranking in a restricted class of functions that can be handled with linear programming techniques. Previous algorithms based on affine rankings either are applicable only to simple loops (i.e., single-node flowcharts) and rely on enumeration, or are not complete in the sense that they are not guaranteed to find a ranking in the class of functions they consider, if one exists. We proposed an efficient algorithm to compute ranking functions, reinvesting most of the techniques from [25] to schedule static loops. It can handle flowcharts of arbitrary structure, the class of candidate rankings it explores is larger, and our method, although greedy, is provably complete. In addition to the termination proof, we showed how to use the ranking functions we generate to get upper bounds for the computational complexity (number of transitions) of the source program, again for flowcharts of arbitrary structure. This estimate is a polynomial, which means that we can handle programs with more than linear complexity. This work has been presented at the SAS'10 conference [5]. As a complement, the survey paper published at MEMOCODE'10 [4] recalls the connection between algorithms for parallelism detection in loops and multi-dimensional scheduling of recurrence equations, and this additional connection with ranking functions for program termination.

We have built a complete software suite, which first uses the `C2fsm` tool to convert the C source into a counter automaton. The Aspic tool is then responsible for computing invariants as polyhedral approximations. Finally, they are given to RanK, which builds a ranking (if any) using Pip. RanK also computes the WCCC using the Ehrhart polynomial module of the Polylib. The first two stages of this toolchain (`C2fsm` and Aspic) have been presented at the workshop TAPAS'10 [11]. Thanks to this toolchain, our method is able to handle every program whose control can be translated into a counter automaton. This roughly covers programs whose control depends on integer variables exclusively, using conditionals, DO loops and WHILE loops whose tests are affine expressions on variables. Furthermore, it is easy to approximate programs which are outside this class by familiar techniques, like ignoring non-affine tests or variables with too complex a behavior. Termination of the approximate program entails termination of the original program, but the converse is not true.

This method can be extended in several interesting directions:

- In some cases, when termination cannot be proved, it is possible to construct a certificate of non-termination in the form of a looping scenario, which should be an invaluable help for debugging.
- The class of ranking functions should be extended to parametric and piecewise affine functions.

6.12. Completeness of instruction selectors

Participant: Florian Brandner.

The use of tree pattern matching for instruction selection has proven very successful in modern compilers. This can be attributed to the declarative nature of tree grammar specifications, which greatly simplifies the development of fast high-quality code generators. The approach has also been adopted widely by generator tools that aim to automatically extract the instruction selector, as well as other compiler components, for application-specific instruction processors from generic processor models. A major advantage of tree pattern matching is that it is suitable for static analysis and allows to verify properties of a given specification. Completeness is an important example of such a property, in particular for automatically generated compilers. Tree automata can be used to prove that a given instruction selector specification is complete, i.e., can actually generate machine code for all possible input programs. Traditional approaches for completeness tests cannot represent dynamic checks that may disable certain matching rules during code generation. However, these dynamic checks occur very frequently in modern compilers and thus need to be modeled in order for a completeness test to be of practical use.

The dynamic checks arise from hidden properties that are not captured by the terminal symbols of the tree grammar notation. We apply terminal splitting to the instruction selector specifications to make these properties explicit. The transformed specification is then verified using a traditional completeness test. If the test fails, counter examples are presented that indicate on which programs the compiler will fail to generate code. We have developed a formal notation of dynamic checks and showed how to perform terminal splitting on an extended form of tree grammars with conditions. The algorithms were implemented in the automatic compiler generator of the xADL processor description language and evaluated using three processor models (MIPS, SPEAR a time-predictable processor developed at VUT, and CHILI a configurable VLIW processor for multimedia processing). Even though terminal splitting increases the problem size for the final completeness test, our experiments show that our method is suited for practical use even in production settings. The test merely required 36 seconds for the huge instruction selector specification of the CHILI processor that consists of more than 1000 translation rules, while the test completes in less than 1.4 second for the other models. An investigation of the use of dynamic checks in existing open source compilers, such as OpenJDK and Quick C - , revealed furthermore that almost all of those could easily be modeled using our approach. This work was presented at the ASAP'10 conference [8].

6.13. Execution models for processors and instructions

Participants: Florian Brandner, Andreas Krall [VUT, Austria], Viktor Pavlu [VUT, Austria].

Modeling the execution of a processor and its instructions is a challenging problem, in particular in the presence of long pipelines, parallelism, and out-of-order execution. A naive approach based on finite state automata inevitably leads to an explosion in the number of states and is thus only applicable to simple minimalistic processors. During their execution, instructions may only proceed *forward* through the processor's datapath towards the end of the pipeline. The state of later pipeline stages is thus independent of potential hazards in preceding stages. This also applies for data hazards, i. e., we may observe data by-passing from a later stage to an earlier one, but not in the other direction.

Based on this observation, we explored the use of a series of *parallel finite automata* (PFA) to model the execution states of the processor's resources individually. The automaton model captures state updates of the individual resources along with the movement of instructions through the pipeline. A highly-flexible synchronization scheme built into the automata enables an elegant modeling of parallel computations, pipelining, and even out-of-order execution. An interesting property of our approach is the ability to model a subset of a given processor using a sub-automaton of the full execution model. The PFA-based processor models are suited for a rich set of applications ranging from accurate representation of processor constraints in compilers, precise timing models for worst-case execution time analysis, verification and validation of processor designs and programs, the synthesis of hardware models, and processor simulation. For example, in cooperation with Viktor Pavlu and Andreas Krall from the Vienna University of Technology work on a tracing-based processor simulation framework using PFA models has been started. First results were published at the NORCHIP'10 conference [9].

7. Contracts and Grants with Industry

7.1. Nano2012 MEDIACOM project with stmicroelectronics on SSA, register allocation, and JIT compilation

Participants: Benoit Boissinot, Florian Brandner, Quentin Colombet, Alain Darte, Fabrice Rastello.

This contract has started in September 2009 as part of the funding mechanism Nano2012. This is the continuation of the successful previous project Sceptre with STMicroelectronics, which ended in December 2009. This new project concerns both aggressive optimizations and the application of the previously-developed techniques to JIT compilation. Related activities are described in Sections 6.3 to 6.8.

7.2. Nano2012 S2S4HLS project with stmicroelectronics on source-to-source transformations for high-level synthesis

Participants: Christophe Alias, Alain Darte, Paul Feautrier, Alexandru Plesco.

This contract has started in January 2009 as part of the funding mechanism Nano2012. This is a joint project with the Cairn Inria project-team and STMicroelectronics, whose goal is the study and development of source-to-source program transformations, in particular loop transformations, that are worth applying on top of HLS tools. This includes restructuring transformations, program analysis, memory optimizations and array reshaping, etc. The work presented in Section 6.9 is part of this project.

8. Other Grants and Activities

8.1. National initiatives

- So far, the french compiler community had no official national meetings. In 2010, Fabrice Rastello has decided to push the different actors in compilation to meet regularly. He contacted all groups whose activities are related to compilation and he organized the first "compilation day" in September 2010 in Lyon. The second session took place in Aussois during 3 days in December 2010. The effort of Fabrice Rastello seems to be a success: the community is now well identified and such an event will occur at least once a year.
- Paul Feautrier and Christophe Alias meet regularly with different actors of the high-level synthesis french community. This working group is intended to lead to an ANR proposal in the future. They been involved in the preparation of a proposal on HLS to be submitted to the ANR Arpege initiative. The partners come from academia (TIMA, IRISA, LasTIC, ASIM) and from industry (Thales, Bull). A first version of the proposal was submitted in 2010, but was rejected mostly on the ground that the project leader should have been from industry rather than academia. A revised version, under the leadership of the Magillem company, is in preparation and will be submitted in March 2011.

8.2. European initiatives

- Compsys has obtained a PROCOPE (France-Germany) funding to collaborate in 2010-2011 with Sebastian Hack's team (Sarrebrücken) on register allocation and aliasing problems. This project led to several 3-days visits. Also, a ENS-LYON student (Amaury Pouly) spent several months in Sebastian Hack's group for his Master 1 training period.
- Paul Feautrier is in regular contact with prof. Christian Lengauer of the University of Passau (Germany), where he spent the month of April 2010 on a grant from BFHZ-CCUFB. Among the outcomes of this stay was the joint writing of an Encyclopedia entry on the Polyhedral Model.

8.3. International initiatives

- Fabrice Rastello has obtained a FRAPEMIG-INRIA (Brazil-France) funding to collaborate with Mariza A. S. Bigonha, Fernando M. Q. Pereira, and Roberto S. Bigonha from the Fereral University of Mina Gerais (UFMG) in Brazil. A student from this group, Andre Tavares, was hosted by Compsys in Spring/Summer 2010.
- Since July 2010, Fabrice Rastello has started a sabbatical year at Colorado State University within the group of Sanjay Rajopadhye, and in connection with the PathScale compiler company.

8.4. Informal cooperations

- Fabrice Rastello and Alain Darte have regular contacts with Jens Palsberg at UCLA (Los Angeles, USA), Sebastian Hack at Saarland University (Saarbrücken, Germany), Philip Brisk at University of California Riverside (Riverside, USA), and Benoit Dupont-de-Dinechin (Kalray, Grenoble).
- Christophe Alias is in regular contact with P. Sadayappan at Ohio State University (USA) and J. (Ram) Ramanujam at Lousiana State University (USA).
- Compsys is in regular contact with Christine Eisenbeis, Albert Cohen and Sid-Ahmed Touati (Inria project Alchemy, Paris), with Steven Derrien and Patrice Quinton (Inria project Cairn, Rennes), with Alain Greiner (Asim, LIP6, Paris), and Frédéric Pétrot (TIMA, Grenoble).
- Compsys, as some other Inria projects, is involved in the network of excellence HiPEAC (High-Performance Embedded Architecture and Compilation, http://www.hipeac.net/). Compsys is also partner of the network of excellence Artist2 to keep an eye on the developments of MPSoC and to disseminate past work on automatic parallelization.
- Florian Brandner is collaborating with the group of Andreas Krall at the Vienna University of Technology on topics related to the processor description language xADL and on compilation for explicitly parallel processors (EPICOpt, http://www.complang.tuwien.ac.at/epicopt/). He is additionally working with Martin Schöberl from the Technical University of Denmark (DTU) on topics evolving around time-predictable computing.
- Alain Darte is in contact with Yann Orlarey from the Grame team (Lyon, “Centre National de Création Musicale”) for a possible collaboration on the development of Faust, a compiled language for real-time audio signal processing.
- In Fall 2010, Compsys hosted Amir Ben Amram (from Tel-Aviv University) for 2 weeks, in a joint organization with the PLUME LIP team (“proofs and languages”), to work on different aspects of program termination and worst-case complexity.

9. Dissemination

9.1. Conferences, journals, and book chapters

- Fabrice Rastello is the local chair of the CGO’11 conference (ACM/IEEE International Conference on Code Generation and Optimization) that will be held on April 2011 in Chamonix, France. Christophe Alias is the program chair and the main organizer of the first international workshop on polyhedral compilation techniques (IMPACT’11) that will be held in conjunction with CGO’11.
- In collaboration with participants of the SSA’09 workshop, Fabrice Rastello is preparing a book on SSA (Static Single Assignment) and its application in compilation, optimization, and program analysis. In a joint work with Diego Novillo (Google), Florian Brandner contributed a chapter on sparse dataflow analysis to this book.

- Alain Darte was member of the program committees of Scopes'10 (International Workshop on Software and Compilers for Embedded Systems), CC'11 (Compiler Construction), and DATE'11 (Design and Test in Europe). He is member of the steering committee of the workshop series CPC (Compilers for Parallel Computing). He is member of the editorial board of the international journal ACM Transactions on Embedded Computing Systems (ACM TECS).
- Paul Feautrier is associate editor of Parallel Computing and the International Journal of Parallel Computing. He is a member of the scientific committee for the Encyclopedia of Parallel Programming, to be published soon by Springer under the direction of prof. David Padua from the University of Illinois. He has contributed four entries [19], [17], [18], [16] and reviewed more than ten entries on related subjects. Alain Darte has contributed one entry [15].

9.2. Teaching and thesis advising

- In 2010, Christophe Alias gave a Master 1 course on "Compilation" at ENS-Lyon, and a L3 course on "Introduction to compilation" at ENSI Bourges. Alain Darte gave a Master 2 course on "Advanced Program Optimizations" at ENS-Lyon. Alexandru Plesco gave TDs and TPs (lab work courses) on "Operating System Design" and "Algorithms" as an ATER (teaching assistant) at INSA-Lyon.
- Christophe Alias, Fabrice Rastello and Florent Dupont de Dinechin were co-organizers of the Winter School "Beyond the PC. Application specific systems: design and implementation" that took place in February 2010 at ENS-Lyon, as part of the Master of ENS-Lyon.
- Fabrice Rastello was thesis advisor of Benoit Boissinot who defended his PhD in September 2010 [1]. Christophe Alias and Alain Darte were thesis co-advisors (also with Tanguy Risset) of Alexandru Plesco who defended his PhD in September 2010 [2]. Alain Darte and Fabrice Rastello are currently co-advisors of the PhD thesis of Quentin Colombet who started in January 2010. Florian Brandner is co-advising the Master thesis of Robert Stefan on extending the type system of the xADL processor description language. Robert is located in Vienna and is also supervised by Viktor Pavlu and Andreas Krall.
- Alain Darte was the vice-president of the 2010 admission exam to ENS-LYON, responsible for the "Computer Science" part.
- Christophe Alias belongs to the teaching council of the Computer Science Department of ENS-LYON.

9.3. PhD defense committees and hiring committees

- Alain Darte was member of the hiring committee of Inria for junior researchers (at Lille) and for an assistant professor position at Ensimag (Verimag laboratory).
- Paul Feautrier was a rewiever for the PhD of Louis-Noël Pouchet (defended January 18, 2010, Paris-Sud) and Alexandre Becoulet (defended September 28, 2010, UPMC). Alain Darte was reviewer for the PhD of Benoit Robillard (defended November 30, 2010, CNAM). He was also member of the "Habilitation thesis" defense of Sid-Ahmed Touati (defended June 30, 2010, UVSQ).

9.4. Workshops, seminars, and invited talks

(For conferences with published proceedings, see the bibliography.)

- Alain Darte gave a talk [13] at CPC'10 (international workshop on Compilers for Parallel Computing) and an invited tutorial [4] at MEMOCODE'10 (ACM/IEEE International Conference on Formal Methods and Models for Codesign).
- Quentin Colombet also gave a talk at CPC'10 [12].
- During research visits to the Technical University of Denmark in Copenhagen and at IRISA (Rennes), Florian Brandner gave presentations of his past and ongoing research activities. He also gave a talk at the second workshop of the french compilation community in Aussois.
- Alexandru Plesco gave a talk at the first and second meetings of the french compilation community, in Lyon and Aussois.

10. Bibliography

Publications of the year

Doctoral Dissertations and Habilitation Theses

[1] B. BOISSINOT. *Towards an SSA-based Compiler Back-end: Some Interesting Properties of SSA and Its Extensions*, École normale supérieure de Lyon, September 2010.

[2] A. PLESCO. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*, École normale supérieure de Lyon, September 2010.

Articles in International Peer-Reviewed Journal

[3] B. BOISSINOT, P. BRISK, A. DARTE, F. RASTELLO. *SSI Properties Revisited*, in "ACM Transactions on Embedded Computing Systems", 2010, Special Issue on Software and Compilers for Embedded Systems, to appear.

Invited Conferences

[4] A. DARTE. *Understanding Loops: The Influence of the Decomposition of Karp, Miller, and Winograd*, in "8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)", Grenoble, France, IEEE Computer Society, July 2010, p. 139-148.

International Peer-Reviewed Conference/Proceedings

[5] C. ALIAS, A. DARTE, P. FEAUTRIER, L. GONNORD. *Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs*, in "17th International Static Analysis Symposium (SAS'10)", Perpignan, France, ACM press, September 2010, p. 117-133.

[6] C. ALIAS, A. DARTE, A. PLESCO. *Optimizing DDR-SDRAM Communications at C-Level for Automatically-Generated Hardware Accelerators. An Experience with the Altera C2H HLS Tool*, in "21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'10)", Rennes, France, IEEE Computer Society, July 2010, p. 329-332.

[7] F. BOUCHEZ, Q. COLOMBET, A. DARTE, C. GUILLON, F. RASTELLO. *Parallel Copy Motion*, in "13th International Workshop on Software & Compilers for Embedded Systems (Scopes'10)", St. Goar, Germany, June 2010.

[8] F. BRANDNER. *Completeness of Automatically Generated Instruction Selectors*, in "21st International Conference on Application-specific Systems Architectures and Processors (ASAP'10)", Rennes, France, IEEE Computer Society, July 2010, p. 175–182.

[9] F. BRANDNER, V. PAVLU, A. KRALL. *Execution Models for Processors and Instructions*, in "28th Norchip Conference (NORCHIP'10)", November 2010.

[10] B. DIOUF, A. COHEN, F. RASTELLO, J. CAVAZOS. *Split Register Allocation: Linear Complexity Without the Performance Penalty*, in "International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC'10)", Lecture Notes in Computer Science, Springer Verlag, January 2010, vol. 5952, p. 66-80.

[11] P. FEAUTRIER, L. GONNORD. *Accelerated Invariant Generation for C Programs with Aspic and C2fsm*, in "Workshop on Tools for Automatic Program Analysis (TAPAS'10)", Electronic Notes in Theoretical Computer Science, September 2010, vol. 267, n^{o} 2, p. 3-13.

Workshops without Proceedings

[12] Q. COLOMBET. *Parallel Copy Motion*, in "Workshop Compilers for Parallel Computing (CPC'10)", Vienna, Austria, July 2010.

[13] A. DARTE. *Optimizing DDR-SDRAM Communications at C-Level for Automatically-Generated Hardware Accelerators. An Experience with the Altera C2H HLS Tool*, in "Workshop Compilers for Parallel Computing (CPC'10)", Vienna, Austria, July 2010.

Research Reports

[14] C. ALIAS, B. PASCA, A. PLESCO. *Automatic Generation of FPGA-Specific Pipelined Accelerators*, Laboratoire de l'Informatique du Parallélisme (LIP), 2010, n^{o} RR2010-37.

Scientific Popularization

[15] A. DARTE. *Scheduling of uniform recurrence equations and optimal parallelism detection in loops*, in "Encyclopedia of Parallel Programming", D. PADUA (editor), Springer, 2010, to appear.

[16] P. FEAUTRIER. *Array Layout for Parallel Processing*, in "Encyclopedia of Parallel Programming", D. PADUA (editor), Springer, 2010, to appear.

[17] P. FEAUTRIER. *Bernstein's Conditions*, in "Encyclopedia of Parallel Programming", D. PADUA (editor), Springer, 2010, to appear.

[18] P. FEAUTRIER. *Dependences*, in "Encyclopedia of Parallel Programming", D. PADUA (editor), Springer, 2010, to appear.

[19] P. FEAUTRIER, C. LENGAUER. *The Polyhedron Model*, in "Encyclopedia of Parallel Programming", D. PADUA (editor), Springer, 2010, to appear.

References in notes

[20] P. BOULET, P. FEAUTRIER. *Scanning Polyhedra without DO loops*, in "PACT'98", October 1998.

[21] A. DARTE, R. SCHREIBER, G. VILLARD. *Lattice-Based Memory Allocation*, in "IEEE Transactions on Computers", October 2005, vol. 54, n^{o} 10, p. 1242-1257, Special Issue: Tribute to B. Ramakrishna (Bob) Rau.

[22] B. DUPONT DE DINECHIN, C. MONAT, F. RASTELLO. *Parallel Execution of Saturated Reductions*, in "Workshop on Signal Processing Systems (SIPS'01)", IEEE Computer Society Press, 2001, p. 373-384.

[23] P. FEAUTRIER. *Scalable and Structured Scheduling*, in "International Journal of Parallel Programming", October 2006, vol. 34, n^{o} 5, p. 459–487.

[24] P. FEAUTRIER. *Dataflow Analysis of Scalar and Array References*, in "International Journal of Parallel Programming", February 1991, vol. 20, n^{o} 1, p. 23–53.

[25] P. FEAUTRIER. *Some Efficient Solutions to the Affine Scheduling Problem, Part II, Multidimensional Time*, in "International Journal of Parallel Programming", December 1992, vol. 21, n^{o} 6.

[26] A. FRABOULET, K. GODARY, A. MIGNOTTE. *Loop Fusion for Memory Space Optimization*, in "IEEE International Symposium on System Synthesis", Montréal, Canada, IEEE Press, October 2001, p. 95–100.

[27] R. JOHNSON, M. SCHLANSKER. *Analysis of Predicated Code*, in "International Workshop on Microprogramming and Microarchitecture (Micro-29)", 1996.

[28] A. STOUTCHININ, F. DE FERRIÈRE. *Efficient Static Single Assignment Form for Predication*, in "International Symposium on Microarchitecture", ACM SIGMICRO and IEEE Computer Society TC-MICRO, 2001.

[29] A. TURJAN, B. KIENHUIS, E. DEPRETTERE. *Translating affine nested-loop programs to process networks*, in "International conference on Compilers, architecture, and synthesis for embedded systems (CASES'04)", New York, NY, USA, ACM, 2004, p. 220–229.

[30] S. VERDOOLAEGE, H. NIKOLOV, N. TODOR, P. STEFANOV. *Improved derivation of process networks*, in "International Workshop on Optimization for DSP and Embedded Systems (ODES'06", 2006.

[31] F. DE DINECHIN, C. KLEIN, B. PASCA. *Generating High-Performance Custom Floating-Point Pipelines*, in "Field Programmable Logic and Applications", IEEE, August 2009, http://prunel.ccsd.cnrs.fr/ensl-00379154/.