INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Compsys

Compilation and Embedded Computing Systems

Grenoble - Rhône-Alpes

Theme : Architecture and Compiling

Activity Report

2009

Table of contents

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-L*yon* UCB-L*yon Inria 5668), located at* ENS-L*yon, and as an Inria pre-project. It became a full Inria project in January 2004. It has been evaluated in Spring 2007, positively, and will continue 4 more years.*

1. Team

Research Scientist

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

Faculty Member

Paul Feautrier [Professor ENS-Lyon, emeritus, HdR]

Technical Staff

Quentin Colombet [Contract Minalogic Sceptre, Nov. 2007-...]

PhD Student

Benoit Boissinot [ENS-Lyon grant, Sept. 2006-...]
Florent Bouchez [ENS-Lyon grant, Sept. 2005-May 2009]
Alexandru Plesco [MESR grant, Sep. 2006-...]

Post-Doctoral Fellow

Florian Brandner [Contract Nano2012 Mediacom, Dec. 2009-...]
Laure Gonnord [ATER UCB-Lyon, Sep. 2008-Jun. 2009]

Administrative Assistant

Caroline Suter [Part-time]

2. Overall Objectives

2.1. Introduction

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 interface, 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 will continue with updated research directions. The next section defines the general context of Compsys activities. The second section specifies 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 upcoming years. The last section highlights the main results of 2009.

2.2. General presentation

Classically, an embedded computer is a digital system, part of a larger system (appliances like phones, TV sets, washing machines, game platforms, or even larger systems like radars and sonars), which is not directly accessible to the user. In particular, this computer is not programmable in the usual way. Its program, if it exists, has been loaded as part of the manufacturing process, and is seldom (or never) modified. Today, as the embedded systems market grows and evolves, this view of embedded systems tend to be too restrictive. Many aspects of general-purpose computers apply to embedded platforms. Nevertheless, embedded systems remain characterized by application types, constraints (cost, power, efficiency, heterogeneity), market. 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) computer science for embedded systems is not well developed in France, especially if one considers the importance of actors like Alcatel, STMicroelectronics, Matra, Thales, etc.; c) since embedded systems have an increasing 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 its 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 central 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. But unlike more standard compilation 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 intermediate platform-independent languages (Java bytecode-type) for which, on the contrary, lighter compilation mechanisms (i.e., faster and less memory-consuming) must be developed for this dynamic/just-in-time compilation. Our objective for compilation for embedded computing systems is to revisit past compilation techniques, to deconstruct them, to improve them, and to develop new techniques taking into account constraints of embedded processors.

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 incidence 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 between 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 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 time performances, power consumption, code and die size, memory constraints, 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.

Before its creation, all members of Compsys have been working, more or less, in the field of automatic parallelization and high-level program transformations. Paul Feautrier was the initiator of the polytope model for program transformations in the 90s 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. Most researchers in France working on high-performance computing (automatic parallelization, languages, operating systems, networks) moved to grid computing at the end of 90s. We thought that applications, industrial needs, and research problems were more important 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.

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 under 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 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 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 2009

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. This work has led to three new developments:

- An algorithm for out-of-SSA conversion that improves previous approaches in all aspects: it is provably correct, more general, easier to implement, faster, and less memory-consuming. This algorithm, developed by Benoit Boissinot, Alain Darte, Fabrice Rastello, and other colleagues from STMicroelectronics, was presented at the IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2009) and received the best paper award (third year in a row for the Compsys team, which seems extremely rare, if not unique, in any conference).
- 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. This work is the result of an informal collaboration of Benoit Boissinot, Alain Darte, and Fabrice Rastello with Philip Brisk from the EPFL, Lausanne.
- A technique for moving parallel copies between registers, particularly useful to be able to apply SSA-based register allocation even to control-flow graphs with “critical edges”.

The Sceptre Minalogic project was finished and (very) positively evaluated at the end of 2009. The first implementation of a full SSA-based register allocator, with two phases (spilling + coloring/coalescing), was done by Quentin Colombet. Florent Bouchez defended his PhD in May 2009 on this topic. Alain Darte and Fabrice Rastello participated to a tutorial on "SSA-based register allocation" at CGO'09. This tutorial was given again at LCPC'09 by Florent Bouchez and Fabrice Rastello. The first (international) workshop focusing on SSA was co-organized by Fabrice Rastello, with the help of Alain Darte and Benoit Boissinot, at Autrans in April 2009. The Sceptre project will continue through Mediacom, a new 3-years project funded by Nano2012 (governmental help for R&D).

In high-level synthesis, two aspects have been explored:

- High-level transformations for improving performances and optimizing communications, in particular by analyzing the industrial tool Altera C2H. A joint project with the Cairn Inria project has started, in collaboration with STMicroelectronics, on source-to-source transformations for high-level synthesis (S2S4HLS), also funded within Nano2012. A collaboration is also in progress with all french specialists on HLS, which should lead to an ANR proposal.
- Program termination and worst-case computational complexity, i.e., briefly speaking an evaluation of the number of iterations in looping processes such as codes with while loops and jumps (abstract worse-case execution time). The initial motivation of this research, which is part of the S2S4HLS project, was to be able to analyze/transform codes with while loops so that they can be accepted by HLS tools. The main result, unexpected because surprising, is that we can reuse, for an apparently-disconnected topic (program termination), all polyhedral tools developed in high-performance computing: our multi-dimensional scheduling techniques and 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.

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 the 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 [17] 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 CLI 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 [23], 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 [24], 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 SSA.
- 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, 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 [1], Gaut [2], UGH [3], CloogVHDL [4]), and in industry (such as C2H [5], CatapultC [6], PICO Express [7]). 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

[1] http://mesl.ucsd.edu/spark
[2] http://www-labsticc.univ-ubs.fr/www-gaut/
[3] http://www-asim.lip6.fr/recherche/disydent/disydent_sect_12.html
[4] http://users.elis.ugent.be/~hmdevos/CLooGVHDL/
[5] http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html
[6] http://www.mentor.com/products/esl/high_level_synthesis/
[7] http://www.synfora.com/products/picoexpress.html

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 [18] and to process networks [25], [26].

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 [19]. 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 problem 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 [22], techniques for data allocation in scratch-pad memory, or techniques for folding multi-dimensional arrays [16]. 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 to 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 initial goal of submitting an ANR proposal at the next opportunity.

4. Application Domains

4.1. Application Domains

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

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

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 [14], 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, post-doc Inria, 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.) Cl@k has been plugged by Christophe Alias into Rose, a source-to-source program transformer, thanks to the development of a lifetime analyzer called Bee. Bee uses Rose, as a high-level 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. 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, Fabrice Baray, Alain Darte.

Bee is – to our knowledge – the only complete tool, from source to source, able to contract arrays. It has been developed by Christophe Alias and represents more than 7000 lines of code. It rewrites a kernel written in C to reduce the size of arrays, bridging the gap between the theoretical framework described in [16] and implemented in Cl@k, and effective program transformations for array contraction. For that, a novel and 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 allowing 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 get 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 function (the corresponding allocations). We currently use this feature in source-to-source transformations for high-level synthesis (see Section 3.3).

As many polyhedral tools, Bee makes an intensive use of the state-of-the-art libraries Pip (parameterized integer programming) and Polylib (polyhedra set operations). The input program is parsed thanks to Rose, a library developed by D. Quinlan at Lawrence Livermore National Labs (USA). Rose provides various features to manipulates ASTs and a unified C++ interface on EDG, an industrial C/C++ parser from Edison group. The robustness of the parser (used in Intel compilers) is another important feature that enlarges the application domain of Bee.

5.6. RanK

Participants: Christophe Alias, Laure Gonnord [Former ATER in Compsys, now MCF Lille], Paul Feautrier, Alain Darte.

RanK is a tool proving the termination of a program (in some cases) by computing a *ranking function*, a mapping from the operations of the program to a well-founded set that *decreases* as the computation advances. In case of success, RanK provides an upper bound of the worst-case time complexity of the program as a symbolic expression involving the input variables of the program (parameters). 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. RanK, implemented by Christophe Alias, represents more than 3000 lines of C++ code. RanK uses several high-level symbolic features developed in Bee. The input of RanK is an integer automaton, representing the control structure of the program to check, which is computed by c2fsm. 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 for many benchmarks programs of the community. It uses the libraries Piplib and Polylib.

5.7. 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.

5.8. LAO developments in aggressive compilation

Participants: Benoit Boissinot, Florent Bouchez, Quentin Colombet, Alain Darte, 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 has developed and integrated into the main trunk of LAO a full implementation of a two-phases register allocation. This 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 6.4).

5.9. LAO developments in JIT compilation

Participants: Benoit Boissinot, Alain Darte, Benoit Dupont-de-Dinechin [STMicroelectronics], 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 (see Section 6.2), and we programmed and tested various ways to compute the liveness information as described in 6.6. Recent efforts (see Section 7.2) also focused on developing a tree-scan register allocator for the JIT part of the compiler. This is partially done. In particular there remains to develop a JIT conservative coalescing. Our goal is to bias our tree-scan coalescing with the result of our JIT aggressive coalescing (see Section 6.2).

6. New Results

6.1. Introduction

This section presents the results obtained by Compsys in 2009. For clarity, some earlier work is also recalled, when results were continued or extended in 2009.

6.2. Revisiting out-of-SSA translation for correctness, efficiency, and speed

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

Static single assignment (SSA) form is an intermediate program representation in which many code optimizations can be performed with fast and easy-to-implement algorithms. However, some of these optimizations create situations where SSA variables arising from the same original variable now have overlapping live ranges. This complicates the translation out of SSA code into standard code. There are three issues: correctness, code quality (elimination of useless copies), and algorithm efficiency (speed and memory footprint). Briggs et al. proposed patches to correct the initial approach of Cytron et al. A cleaner and more general approach was then proposed by Sreedhar et al., along with techniques to reduce the number of generated copies. We went one step further. We proposed a conceptually simpler approach, based on coalescing and a more precise view of interferences, in which correctness and optimizations are separated. Our approach is provably correct and simpler to implement, with no patches or particular cases as in previous solutions, while reducing the number of generated copies. Also, experiments with SPEC CINT2000 show that it is 2x faster and 10x less memory-consuming than the Method III of Sreedhar et al., which makes it suitable for just-in-time compilation.

These results have been presented at CGO'09 [6] and were acknowledged by the **best paper award**.

6.3. 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 will be presented at the HiPEAC'10 conference [7].

6.4. Register-allocated SSA destruction: handling of critical edges

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

Recent results on the SSA form opened promising directions for the design of new register allocation heuristics for JIT compilation. In particular, heuristics based on tree scans with two decoupled phases, one for spilling, one for splitting/coloring/coalescing, seem good candidates for designing memory-friendly, fast, and competitive register allocators. 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 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.

6.5. "Optimal" formulation of register spilling

Participants: Alain Darte, Quentin Colombet, Fabrice Rastello.

This work is part of the contract (see Section 7.2) with the CEC team at STMicroelectronics. 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.

6.6. Fast computation of liveness sets

Participants: Benoit Boissinot, 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 variables in 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 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. In SSA, 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 the liveness sets, one variable at a time, with the adequate data structures.

Finally, we demonstrated and compared the efficiency of our solutions, for different data structures of liveness sets (bitsets and ordered sets), on the Spec CPU2006 programs in a production compiler. For our experiments, our solutions outperform standard data-flow liveness analysis, as we could expect. The two-passes data-flow algorithm is, in general, a bit slower than the algorithm that considers one variable at a time. More surprising, this latter algorithm can be easily extended to compute liveness sets for standard (i.e., non-SSA form) programs and is still faster than standard iterative data-flow liveness analysis. Nevertheless, some work remains to be done to finalize this preliminary experimental study.

6.7. 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 is 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.). Our paper corrects some of the mistakes that have been made. 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. Our proof is based on the notion of loop nesting forest.

This work is the result of an informal collaboration with Philip Brisk from the University of Lausanne (EPFL).

6.8. Loop transformations for high-level synthesis and communication optimizations

Participants: Christophe Alias, Alain Darte, Alexandru Plesco, Tanguy Risset [Prof. Insa-Lyon].

We are studying the use of source-to-source loop transformations performed as a front-end to high-level synthesis (HLS) tools. Our first study was based on the Wrapit loop transformation tools developed by the Alchemy team project and integrated into the ORC open-source compiler. The Wrapit tool has been applied to C code, synthesized by the Spark HLS tool, showing important performance improvements of the resulting design (in terms of silicon area and communication optimization). This work was done by Alexandru Plesco and presented at SYMPA'08.

Despite the resulting improvements, this first study revealed the difficulty of interfacing HLS designs with external components (host processor or memory). The Spark HLS tool, as many other HLS tools, is indeed not able to analyze or optimize data transfers or storage for them. More generally, these results confirm that there is a strong need for data communication/storage mechanisms between the host processor and the hardware accelerator. Alexandru Plesco is currently investigating how to design a hardware/software interface model for enabling communication optimizations, such as burst communications. We use Altera C2H, a more advanced HLS tool than Spark, which is able to communicate not only through pins or FIFOs, but also through arrays, with pipelined access to external memories.

So far, the results are very promising, despite the extreme difficulty to use and "control" the behavior of HLS tools. By mixing tiling transformations and the generation of communicating processes, we can improve the communication bandwidth to a DDR memory by a factor of more than 10 on the simple examples we considered. We still have to automate the complete transformation chain, using our program transformer Bee, and to handle more complex examples.

6.9. Program termination and worst-case computational complexity

Participants: Christophe Alias, Laure Gonnord [Former ATER in Compsys, now MCF Lille], 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 [21], i.e., a function from the program states to a well-founded set, which strictly decreases at each program step. A standard method to automatically generate such a function is to compute invariants)approximation of all possible values of program variables) for each program point and to search 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.

Our first contribution is to propose an efficient algorithm to compute ranking functions, reinvesting most of the techniques from [20] 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. 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 constructs a ranking (if any) using Pip. 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.

Our second contribution is to show 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. RanK computes the WCCC using the Ehrhart polynomial module of the Polylib.

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.

7. Contracts and Grants with Industry

7.1. Minalogic SCEPTRE project with STMicroelectronics on SSA, register allocation, and aggressive compilation

Participants: Benoit Boissinot, Florent Bouchez, Quentin Colombet, Alain Darte, Fabrice Rastello.

This contract started in October 2006 as part of the "pôle de compétitivité" MINALOGIC. The collaboration deals with the possible applications of the SSA form for program optimizations. It concerns predication, register allocation, and instruction selection. Related results are described in Sections 6.2 and 6.4, as well as in previous annual reports.

7.2. 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 a continuation of the successful previous project (see Section 7.1) with STMicroelectronics. In particular, it concerns the application of the previously-developed techniques to JIT compilation. Related activities are described in Sections 6.5, 6.6, and 6.7.

7.3. 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 whose goal is the study and development of source-to-source program transformations, in particular loop transformtions, that are worth applying before using HLS tools. This includes restructuring transformations, program analysis, memory optimizations and array reshaping, etc.

8. Other Grants and Activities

8.1. 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).
- Compsys applied for a PROCOPE (French-Germany) funding to collaborate with Sebastian Hack's team.
- Fabrice Rastello, Alain Darte, and Quentin Colombet have a FRAPEMIG-INRIA (Brazil-France) funding to collaborate with Dr. Mariza A. S. Bigonha, Dr. Fernando M. Q. Pereira, and Dr. Roberto S. Bigonha from the Fereral University of Mina Gerais (UFMG) in Brazil.

- Compsys is in contact with Francky Catthoor's team in Leuwen (Belgium), with Ed Depreterre's team at Leiden University (the Netherlands), with Peter Marwedel's team in Dortmund. They have participated to joint discussions in the workshop Map2MPSoC of the network of excellence Artist2.
- Alain Darte has fruitful relations with Rob Schreiber at HP Labs, with three joint patents and many publications. The last patent was accepted in 2008 [15].
- Christophe Alias is in contact with Prof. P. Sadayappan at Ohio State University (USA) and Prof. J. (Ram) Ramanujam at Lousiana State University (USA). They participate to a joint work on automatic vectorization.
- 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 disseminate past work on automatic parallelization.

9. Dissemination

9.1. Introduction

This section lists the various scientific and teaching activities of the team in 2009.

9.2. Conferences and journals

- In 2009, Alain Darte was member of the program committees of Scopes'09, CASES'09, Euro-Par'09, and of the new session "fun ideas and thoughts" (FIT) at PLDI'09. 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).
- In 2009, Fabrice Rastello was member of the program committees of the conference CGO'09 (ACM/IEEE Symposium on Code Generation and Optimization). With the help of Alain Darte and Benoit Boissinot, he was the main organizer of SSA'09, the first international workshop on static single assignment (SSA) that has regrouped in April 2009 55 people during 4 days (see http://www.prog.uni-saarland.de/ssasem/). He will also be the local chair of CGO'11 that will be held in Chamonix, France.
- Paul Feautrier is associate editor of Parallel Computing and the International Journal of Parallel Computing. He was a member of the program committee of CC'09 (Compiler Construction) and of the jury for the ASTI 2009 doctoral award.
- Christophe Alias, Fabrice Rastello and Florent Dupont de Dinechin are co-organizers of the Winter School "Beyond the PC. Application specific systems: design and implementation" that will be held in February 2010 at ENS de Lyon.

9.3. Teaching and thesis advising

- In 2009, Fabrice Rastello and Alain Darte gave a Master 2 course on advanced compilation at ENS-Lyon. Christophe Alias and Paul Feautrier both gave a Master 1 course on "Compilation" at ENS-Lyon. Paul Feautrier has been responsible for the "Compilation" project at the L3 level.

- Alain Darte and Fabrice Rastello were thesis co-advisors of Florent Bouchez who defended his PhD in April 2009. Fabrice Rastello is thesis advisor of Benoit Boissinot. Alain Darte and Tanguy Risset are thesis co-advisors of Alexandru Plesco. Christophe Alias participates to this advising too. Alain Darte and Fabrice Rastello will be co-advisors of Quentin Colombet (currently an engineer in Compsys until end of 2009).

9.4. Teaching responsibilities

- Alain Darte is the vice-president of the admission exam to ENS-Lyon, responsible for the "Computer Science" part.
- Christophe Alias belongs to the teaching council of ENS-Lyon.

9.5. Animation

- Paul Feautrier is a member of the PhD committee of LIP, and has been a member the hiring committees of ENS-Lyon for Computer Science and Computational Biology.
- Alain Darte was member of the hiring committees of Inria for junior researchers (Lille) and for the professor position in computer science at ENS-Lyon.
- Alain Darte is the scientific expert, with Alain Girault, in the Inria group in charge of coordinating the joint research efforts of Inria and STMicroelectronics.

9.6. Defense committees

- Paul Feautrier was a rewiever for the PhD of Patrice Gérin (defended November 30, 2009, INPG) and of Louis-Noël Pouchet (to be defended January 18, 2010, Paris-Sud).

9.7. Workshops, seminars, and invited talks

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

- Fabrice Rastello and Alain Darte gave a tutorial on SSA-based Register Allocation at CGO'09 (international conference on Code Generation and Optimization). Fabrice Rastello did it again with Florent Bouchez at LCPC'09 (international workshop on Languages and Compilers for Parallel Computing).
- Alain Darte and Benoit Boissinot both gave a talk at SSA'09 (international workshop on Static Single Assignment). Alain Darte gave a talk at CPC'09 (international workshop on Compilers for Parallel Computing).
- Paul Feautrier was a keynote speaker at LCPC'09 (U. of Delaware, USA).

10. Bibliography

Year Publications

Doctoral Dissertations and Habilitation Theses

[1] F. BOUCHEZ. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*, École normale supérieure de Lyon, April 2009, Ph. D. Thesis.

Articles in International Peer-Reviewed Journal

[2] P. GROSSE, Y. DURAND, P. FEAUTRIER. *Methods for Power Optimization in SOC-Based Data Flow Systems*, in "ACM Transactions on Design Automation of Electronic Systems", vol. 14, n^{o} 3, 2009, p. 1-20, http://doi.acm.org/10.1145/1529255.1529260.

Invited Conferences

[3] F. BOUCHEZ, F. RASTELLO. *SSA-Based Register Allocation*, in "The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09), Newark, Delaware", October 2009, Tutorial with P. Brisk, S. Hack, and J. Palsberg.

[4] A. DARTE, F. RASTELLO. *SSA-Based Register Allocation*, in "International Symposium on Code Generation and Optimization (CGO'09), Seattle", March 2009, Tutorial with P. Brisk and J. Palsberg.

[5] P. FEAUTRIER. *The Polytope Model, Past, Present, Future*, in "The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09), Newark, Delaware", October 2009, Keynote address.

International Peer-Reviewed Conference/Proceedings

[6] B. BOISSINOT, A. DARTE, B. DUPONT DE DINECHIN, C. GUILLON, F. RASTELLO. *Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency*, in "International Symposium on Code Generation and Optimization (CGO'09), Seattle, WA, USA", IEEE Computer Society Press, March 2009, p. 114–125, Best paper award.

[7] 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, to appear US .

[8] O. LABBANI, P. FEAUTRIER, E. LENORMAND, M. BARRETEAU. *Elementary Transformation Analysis for Array-OL*, in "ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'09), Rabat, Morocco", May 2009, p. 362-367.

[9] Q. LU, C. ALIAS, U. BONDHUGULA, S. KRISHNAMOORTHY, J. RAMANUJAM, A. ROUNTEV, P. SADAYAPPAN, Y. CHEN, H. LIN, T.-F. NGAI. *Data Layout Transformation for Enhancing Locality on NUCA Chip Multiprocessors*, in "International ACM/IEEE Conference on Parallel Architectures and Compilation Techniques (PACT'09)", September 2009 US .

[10] M. RASTELLO, F. RASTELLO, H. BELLOT, F. OUSSET, F. DUFOUR. *Size of Snow Particles in a Powder-Snow Avalanche*, in "ASME Fluids Engineering Division Summer Meeting 2009 (FEDSM'09)", August 2009.

Workshops without Proceedings

[11] B. BOISSINOT. *Fast Liveness Checking in SSA*, in "SSA form seminar, Autrans", April 2009.

[12] A. DARTE. *Out-of-SSA Translation*, in "Workshop Compilers for Parallel Computing, Zurich, Switzerland", January 2009.

[13] A. DARTE. *Out-of-SSA Translation*, in "SSA form seminar, Autrans", April 2009.

References in notes

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

[15] A. DARTE, R. SCHREIBER. *System and Method of Optimizing Memory Usage with Data Lifetimes*, April 2008, US patent number 7363459.

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

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

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

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

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

[21] R. W. FLOYD. *Assigning Meaning to Programs*, in "Proc. Symp. on Applied Mathematics", J. T. SCHWARTZ (editor), vol. 19, AMS, 1967.

[22] 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.

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

[24] 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.

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

[26] S. VERDOOLAEGE, H. NIKOLOV, N. TODOR, P. STEFANOV. *Improved derivation of process networks*, in "Proceedings of the 4th International Workshop on Optimization for DSP and Embedded Systems (ODES'06", 2006.