INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Team CAMUS

Compilation pour les Architectures MUlti-coeurS

Nancy - Grand Est

Theme : Architecture and Compiling

Activity Report

2010

Table of contents

1. Team

Faculty Members

Philippe Clauss [Team leader, Professor, Université de Strasbourg, HdR]
Éric Violard [Associate Professor, Université de Strasbourg, HdR]
Vincent Loechner [Associate Professor, Université de Strasbourg]
Alain Ketterlin [Associate Professor, Université de Strasbourg]
Julien Narboux [Associate Professor, Université de Strasbourg]
Nicolas Magaud [Associate Professor, Université de Strasbourg]

PhD Students

Benoît Pradelle [Université de Strasbourg]
Alexandra Jimborean [Université de Strasbourg]
Martin Rouaux [co-supervision, University of Buenos Aires, Argentina]

2. Overall Objectives

2.1. Overall Objectives

The CAMUS team is focusing on developping, adapting and extending automatic parallelizing and optimizing techniques, as well as proof and certification methods, for the efficient use of current and future multicore processors.

The team's research activities are organized into five main issues that are closely related to reach the following objectives: performance, correction and productivity. These issues are: static parallelization and optimization of programs (where all statically detected parallelisms are expressed as well as all "hypothetical" parallelisms which would be eventually taken advantage of at runtime), profiling and execution behavior modeling (where expressive representation models of the program execution behavior will be used as engines for dynamic parallelizing processes), dynamic parallelization and optimization of programs (such transformation processes running inside a virtual machine), object-oriented programming and compiling for multicores (where object parallelism, expressed or detected, has to result in efficient runs), and finally program transformations proof (where the correction of many static and dynamic program transformations has to be ensured).

2.2. Highlights

The CAMUS project reveived positive opinion from the Nancy Grand-Est Comity of Project for the creation of the project-team.

3. Scientific Foundations

3.1. Research directions

The various objectives we are expecting to reach are directly related to the search of adequacy between the sofware and the new multicore processors evolution. They also correspond to the main research directions suggested by Hall, Padua and Pingali in [41]. Performance, correction and productivity must be the users' perceived effects. They will be the consequences of research works dealing with the following issues:

- Issue 1: Static parallelization and optimization
- Issue 2: Profiling and execution behavior modeling
- Issue 3: Dynamic program parallelization and optimization, virtual machine
- Issue 4: Object-oriented programming and compiling for multicores
- Issue 5: Proof of program transformations for multicores

Efficient and correct applications development for multicore processors needs stepping in every application development phase, from the initial conception to the final run.

Upstream, all potential parallelism of the application has to be exhibited. Here static analysis and transformation approaches (issue 1) must be processed, resulting in a *multi-parallel* intermediate code advising the running virtual machine about all the parallelism that can be taken advantage of. However the compiler does not have much knowledge about the execution environment. It obviously knows the instruction set, it can be aware of the number of available cores, but it does not know the effective available resources at any time during the execution (memory, number of free cores, etc.).

That is the reason why a "virtual machine" mechanism will have to adapt the application to the resources (issue 3). Moreover the compiler will be able to take advantage only of a part of the parallelism induced by the application. Indeed some program information (variables values, accessed memory adresses, etc.) being available only at runtime, another part of the available parallelism will have to be generated on-the-fly during the execution, here also, thanks to a dynamic mechanism.

This on-the-fly parallelism extraction will be performed using speculative behavior models (issue 2), such models allowing to generate speculative parallel code (issue 3). Between our behavior modeling objectives, we can add the behavior monitoring, or profiling, of a program version. Indeed current and future architectures complexity avoids assuming an optimal behavior regarding a given program version. A monitoring process will allow to select on-the-fly the best parallelization.

These different parallelizing steps are schematized on figure 1.

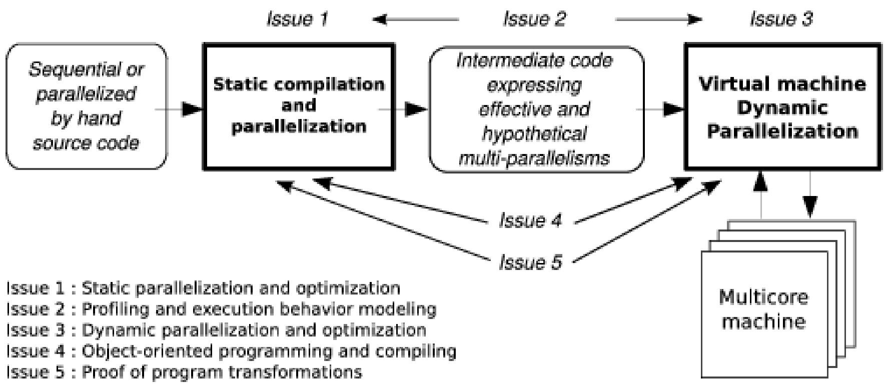
Issue 1
Issue 2
Issue 3
Sequential or parallelized by hand source code
Static compilation and parallelization
Intermediate code expressing effective and hypothetical multi-parallelisms
Virtual machine Dynamic Parallelization
Issue 4
Issue 5
Multicore machine
Issue 1 : Static parallelization and optimization
Issue 2 : Profiling and execution behavior modeling
Issue 3 : Dynamic parallelization and optimization
Issue 4 : Object-oriented programming and compiling
Issue 5 : Proof of program transformations

Figure 1. Automatic parallelizing steps for multicore architectures

The more and more widespread usage of object-oriented approaches and languages emphasizes the need for specific multicore programming tools. The object and method formalism implies specific execution schemes that translate in the final binary by quite distant elementary schemes. Hence the execution behavior control is far more difficult. Analysis and optimization, either static or dynamic, must take into account from the outset this distortion between object-oriented specification and final binary code: how can object or method parallelization be translated (issue 4).

Our project lies on the conception of a production chain for efficient execution of an application on a multicore architecture. Each link of this chain has to be formally verified in order to ensure correction as well as efficiency. More precisely, it has to be ensured that the compiler produces a correct intermediate code, and that the virtual machine actually performs the parallel execution semantically equivalent to the source code: every transformation applied to the application, either statically by the compiler or dynamically by the virtual machine, must preserve the initial semantics. They must be proved formally (issue 5).

In the following, those different issues are detailed while forming our global and long term vision of what has to be done.

3.2. Static parallelization and optimization

Participants: Vincent Loechner, Philippe Clauss, Éric Violard, Alexandra Jimborean.

Static optimizations, from source code at compile time, benefit from two decades of research in automatic parallelization: many works address the parallelization of loop nests accessing multi-dimensional arrays, and these works are now mature enough to generate efficient parallel code [23]. Low-level optimizations, in the assembly code generated by the compiler, have also been extensively dealt for single-core and require few adaptations to support multicore architectures. Concerning multicore specific parallelization, we propose to explore two research directions to take full advantage of these architectures. They are described below.

3.2.1. State of the art

Upstream, an easy interprocedural dependence analysis allows to handle complete programs (but recursivity: recursive functions must be transformed into iterative functions). Concerning iterative control we will use the polyhedral model, a formalism developped these last two decades, which allows to represent the execution of a loop nest by scanning a polytope.

When compiling an application, if it contains loop nests with affine bounds accessing scalars or arrays accessed using affine functions, the polyhedral model allows to:

- compute the dependence graph, which describes the order in which the dependent instructions must be executed [32];
- generate a schedule, which extracts some parallelism from the dependence graph [33], [34];
- generate an allocation, which assigns a processor (or a core) to a set of iterations of the loop nest to be scanned.

This last allocation step needs a thorough knowledge of the target architecture, as many crucial choices will result in performance hazards: for example, the volume and flow of inter-processor communications and synchronization; the data locality and the effects of the TLB (Translation Lookaside Buffer) and the various cache levels and distributions; or the register allocation optimizations. There are many techniques to control these parameters, and each architecture needs specific choices, of a valid schedule, of a parallel loop iterations distribution (bloc-, cyclic-, or tiled), of a loop-unrolling factor, as well as a memory data layout and a prefetch strategy (when available). They require powerfull mathematical tools, such as counting the number of integer points contained in a parametric polytope.

Our own contributions in this area are significant. Concerning schedule and data placement, we proposed new advances in minimizing the number of communications for parallel architectures [51] and in cache access optimizations [50] [8]. We also proposed essential advances in parametric polytope manipulation [9], [5], developped the first algorithm to count integer points in a parametric polytope as an Ehrhart polynomial [3], and proposed successive improvements of this algorithm [10] [62]. We implemented these results in the free software *PolyLib*, utilized by many researchers around the world.

3.2.2. Adapting parallelization to multicore architecture

The first research direction to be explored is multicore specific efficient optimizations. Indeed, multicore architectures need specific optimizations, or we will get underlinear accelerations, or even decelerations. Multicore architectures may have the following properties: specific memory hierarchy, with distributed low-level cache and (possibly semi-) shared high level caches; software-controlled memory hierarchies (memory hints, local stores or scratchpads for example); optimized access to contiguous memory addresses or to separate memory banks; SIMD or vectorial execution in groups of cores, and synchronous execution; higher register allocation pressure when several threads use the same hardware (as in GPGPUs for example); etc.

A schedule and an allocation must be chosen wisely in order to obtain good performances. On NVIDIA GPGPUs, using the CUDA language, Baskaran et al. [22] obtained interesting results that have been implemented in their PLuTo compiler framework. However, they are based on many empirical and imprecise techniques, and require simulations to fine-tune the optimizations: they can be improved. Memory hierarchy efficient control is a cornerstone of tomorrow's multicore architectures performance. Compiler-optimizers have to evolve to meet this requirement.

Simulation and (partial-) profiling may however remain necessary in some cases, when static analysis reaches its intrinsic limits: when the execution of a program depends on dynamic parameters, when it uses complex pointer arithmetic, or when it performs indirect array accesses for example (as is often the case in *while* loops, out of the scope of the classical polyhedral model). In these cases, the compiler should rely on the profiler, and generate a code that interacts with the dynamic optimizer. This is the link with issues 2 and 3 of this research project.

3.2.3. Expressing many potential parallelisms

The dynamic optimizer (issue 3) must be able to exploit various parallel codes to compare them and the best one to choose, possibly swapping from a code to another during execution. The compiler must therefore generate different potentially efficient versions of a code, depending on fixed parameters such as the schedule or the data layout, and dynamic parameters such as the tile size or the unrolling factor.

The compiler then generates many variants of *effective* parallelism, formally proved by the static analyzer. It may also generate variants of code that have not been formally validated, due to the analyzer limits, and that have to be checked during execution by the dynamic optimizer: *hypothetical* parallelism. Hypothetical parallelism could be expressed as a piece of code, valid under certain conditions. Effective and hypothetical parallelisms are called *potential parallelism*. The variants of potential parallelism will be expressed in an intermediate language that has to be discovered.

Using compiler directives is an interesting way to define this intermediate language. Among the usual directives, we distinguish schedule directives for shared memory architectures (such as the OpenMP[1] *parallel* directive), and placement directives for distributed memory architectures (for example the HPF[2] *ALIGN* directive). These two types of directives are conjointly necessary to take full profit of multicore architectures. However, we have to study their complementarity and solve the interdependence or conflict that may arise between them. Moreover, new directives should allow to control data transfers between different levels of the memory hierarchy.

We are convinced that the definition of such a language is required in the next advances in compilation for multicore architectures, and there does not exist such an ambitious project to our knowledge. The OpenCL project[3], presented as an general-purpose and efficient multicore programming environment, is too low-level to be exploitable. We propose to define a new high level language based on compilation directives, that could be used by the skilled programmer or automatically generated by a compiler-optimizer (like OpenMP, recently integrated in the *gcc* compiler suite).

3.3. Profiling and execution behavior modeling

Participants: Alain Ketterlin, Philippe Clauss, Benoît Pradelle.

The increasing complexity of programs and hardware architectures makes it ever harder to characterize beforehand a given program's run time behavior. The sophistication of current compilers and the variety of transformations they are able to apply cannot hide their intrinsic limitations. As new abstractions like transactional memories appear, the dynamic behavior of a program strongly conditions its observed performance. All these reasons explain why empirical studies of sequential and parallel program executions have been considered increasingly relevant. Such studies aim at characterizing various facets of one or several program runs,

[1] http://www.openmp.org
[2] http://hpff.rice.edu
[3] http://www.khronos.org/opencl

e.g., memory behavior, execution phases, etc. In some cases, such studies characterize more the compiler than the program itself. These works are of tremendous importance to highlight all aspects that escape static analysis, even though their results may have a narrow scope, due to the possible incompleteness of their input data sets.

3.3.1. Selective profiling and interaction with the compiler

In its simplest form, studying a given program's run time behavior consists in collecting and aggregating statistics, *e.g.*, counting how many times routines or basic blocks are executed, or counting the number of cache misses during a certain portion of the execution. In some cases, data can be collected about more abstract events, like the garbage-collector frequency or the number and sizes of sent and received messages. Such measures are relatively easy to obtain, are frequently used to quantify the benefits of some optimization, and may suggest some way to improve performance. These techniques are now well-known, but mostly for sequential programs.

These global studies have often been complemented by local, targeted techniques focused on some program portions, *e.g.*, where static techniques remain inconclusive for some fixed duration. These usages of profiling are usually strongly related to the optimization they complement, and are set up either by the compiler or by the execution environment. Their results may be used immediately at run time, in which case they are considered a form of run time optimization [1]. They can also be used offline to provide hints to a subsequent compilation cycle, in which case they constitute a form of profile-guided compilation, a strategy that is common in general purpose compilers.

For instance, in the context where a set of possible parallelizations have been provided by the compiler (see issue 1), a profiling component can easily be made responsible for testing some relevant condition at run time (*e.g.*, that depends on input data) and for selecting the best between various versions of the code. Beyond such simple tasks, we expect that profiling will, at the beginning of the execution, have enough resources to conduct more elaborate analyzes. We believe that combining an "open" static analysis with an integrated profiling component is a promising approach, first because it may relieve the programmer of a large part of the tedious task of implementing the distribution of computations, and second to free the compiler of the obligation to choose between several optimizations in the absence of enough relevant data. The main open question here is to define precisely the respective roles of the compiler and the profiler, and also the amount and nature of information the former can transmit to the latter.

3.3.2. Profiling and dynamic optimization

In the context of dynamic optimization, that is, when the compiler's abilities have been exhausted, a profiler can still do useful work, provided some additional capabilities [1]. If it is able to instrument the code the way, *e.g.*, a PIN-tool does [52], it has access to the whole program, including libraries (or, for example, the code of a low-level library called from a scripting language). This means that it has access to portions of the program that were not under the compiler's control. The profiler can then perform dynamic inter-procedural analyzes, for instance to compute dependencies to detect parallelism that wasn't apparent at compile time because of a function call in the body of a loop. More generally, if the profiler is able to reconstruct at run time some representation of the whole program, as in [71] for example, it is possible to let it search for any construct that can be optimized and/or parallelized in the context of the current execution. Several virtual machines, *e.g.*, for Java or Microsoft CLR, have opened this way of optimizing programs, probably because virtual machines need to maintain an intermediate, structured representation of the running program.

The possibility of running programs on architectures that include a large number of computing cores has given rise to new abstractions [69], [43], [27]. Transactional memories, for instance, aim at simplifying the management of conflicting concurrent accesses to a shared memory, a notoriously difficult problem [45]. However, the performance of a transaction-based application heavily depends on its dynamic behavior, and too many conflicting accesses and rollbacks, severely affect performance. We bet that the need for multicore specific programming tools will lead to other abstractions based on speculative execution. Because of the very nature of speculation, all these abstractions will require run time evaluation, and maybe correction, to avoid pathological cases. The profiler has a central role here, because it can be made responsible for diagnosing inefficient use of speculative execution, and for taking corrective action, which means that it has to be integrated to the execution environment. We also think that the large scope and almost infinite potential uses of a profiling component may well suggest new parallel program abstractions, specially targeted at run time evaluation and adaptation.

3.3.3. Run time program modeling

When profiling goes beyond simple aggregation of counts, it can, for example, sample a program's behavior and split its execution into phases. These phases may help target a subsequent evaluation on a new architecture [63]. When profiling instruments the whole program to obtain a trace, *e.g.*, of memory accesses, it is possible to use this trace for:

- simulation, *e.g.*, by varying the parameters of the memory hierarchy,
- for modeling, *e.g.*, to reconstruct some specific model of the program [71], or to extract dynamic dependencies that help identifying parallel sections [59].

Handling such large execution traces, and especially compressing them, is a research topic by itself [28], [54]. Our contribution to this topic [7] is unusual in that the result of compression is a sequence of loop nests where memory accesses and loop bounds are affine functions of the enclosing loop indices. Modeling a trace this way leads to slightly better average compression rates compared to other, less expressive techniques. But more importantly, it has the advantage to provide a result in symbolic form, and this result can be further analyzed with techniques usually restricted to the static analysis of source code. We plan to apply, in the short term, similar techniques to the modeling of dynamic dependencies, so as to be able to automatically extract parallelism from program traces.

This kind of analysis is representative of a new kind of tools than could be named “parallelization assistants” [49], [59]. Properties that can't be detected by the compiler but that appear to hold in one or several executions of a program can be submitted to the programmer, maybe along a suitable reformulation of its program using some class of abstraction, *e.g.*, compiler directives. The goal is to provide help and guidance in adapting source code, in the same way a classical profiling tool helps pinpoint performance bottlenecks. Control and data dependencies are fundamental to such a tool. An execution trace provides an observed reality; for example a trace of memory addresses. If the observed dynamic dependencies provide a set of constraints, they also suggest a complete family of potential correct executions, be they parallel or sequential, and all these executions are equivalent to the reference execution. Being able to handle large traces, and representing them in some manageable way, means being able to highlight medium to large grain parallelism, which is especially interesting on multicore architectures and often difficult for compilers to discover, for example because of the use of pointers and the difficulty of eliminating potential aliasing. This can be seen as a machine learning problem, where the goal is to recover a hidden structure from a large sequence of events. This general problem has various incarnations, depending on how much the learner knows about the original program, on the kind of data obtained by profiling, on the class of structures sought, and on the objectives of the analysis. We are convinced that such studies will enrich our understanding of the behavior of programs, and of the programming concepts that are really useful. It will also lead to useful tools, and will open up new directions for dynamic optimization.

3.4. Dynamic parallelization and optimization, virtual machine

Participants: Philippe Clauss, Alain Ketterlin, Vincent Loechner, Benoît Pradelle, Alexandra Jimborean.

This link in the programming chain has become essential with the advent of the new multicore architectures. Still being considered as secondary with mono-core architectures, dynamic analysis and optimization are now one of the keys for controling those new mechanisms complexity. From now on, performed instructions are not only dedicated to the application functionalities, but also to its control and its transformation, and so in its own interest. Behaving like a computer virus, such a process should rather be qualified as a “vitamin”. It perfectly knows the current characteristics of the execution environment and owns some qualitative information thanks to a behavior modeling process (issue 2). It appends a significant part of optimizing ability compared to a static compiler, while observing live resources availability evolution.

3.4.1. State of the art

Dynamic analysis and optimization, that is to say simultaneous to the program execution, have motivated a growing interest during the last decade, mainly because of the hardware architectures and applications growing complexity. Indeed, it has become more and more difficult to anticipate any program run simply from its source code, either because its control structures introduce some unknown objects before run (dynamic memory allocation, pointers, ...), or because the interaction between the target architecture and the program generates unpredictable behaviors. This is notably due to the appearance of more optimizing hardware units (prefetching units, speculative processing, code cache, branch prediction, etc.). With multicore architectures, this interest is growing even more. Works achieved in this area for mono-core processors have permitted to establish some classification of the so-called dynamic approaches, either based on the used methodologies or on the objectives.

The first objective for any dynamic approach is to extract some live information at runtime relying on a profiling process. This essential step is the main objective of issue 2 (see sub-section 3.3).

Identifying some "hotspots" thanks to profiling is then used for performance improvement optimizations. Two main approaches can be distinguished:

- the *profile-guided* approach, where analysis and optimization of profile information are performed off-line, that is to say statically. A first run is only performed to extract information for driving a re-compilation. Related to this approach, *iterative compilation* consists in running a code that has been transformed following different optimization possibilities (nature and sequencing of the applied optimizations), and then in re-compiling the transformed code guided by the collected performance information, and so on until obtaining a "best" program version. In order to promote a rapid convergence towards a better solution, some heuristics or some machine learning mechanisms are used [18], [58], [57]. The main drawback of such approaches relates to the quality of the generated code which depends on the reference profiled execution, and more precisely on the used input data set, but also on the used hardware.
- the *on-the-fly* approach consists in performing all steps at each run (profiling, analysis and transformation). The main constraint of this approach is that the time overhead has to be widely compensated by the benefits it generates. Several works propose such approaches dedicated to specific optimizations. We personally successfully implemented a dynamic data prefetching system for the Itanium processor [1].

Although all these works provided some efficient dynamic mechanisms, their adaptation to multicore architectures yields difficult issues, and even challenges them. It is indeed necessary to control interactions between simultaneous tasks, imposing an additional complexity level which can be fateful for a dynamic system, while becoming too costly in time and space.

Some dynamic parallelizing techniques have been proposed in the last years. They are mainly focusing on parallelizing loop-nests, as programs generally spend most of their execution time in iterative structures.

The LRPD test [61] is certainly one of the foundation strategies. This method consists in speculatively parallelizing loops. Privatization and reduction transformations are applied to promote a successful application of the strategy. During execution, some tests are performed to verify the speculation validity. In case of invalid speculation, the targeted loop is re-executed sequentially. However, the application range is limited to loops accessing arrays; pointers cannot be handled. Moreover the method is not fully dynamic since an initial static analysis is needed.

In [31], Cintra and Llanos present a speculative parallel execution mechanism for loops, where iteration chunks are executed in sliding windows of n threads. The loops are not transformed and the sequential schedule remains as a reference to define a total order on the speculative threads. In order to verify whether some dependencies are violated during the program run, all data structures qualified as speculative, that is to say those being accessed in read-write mode by the threads, are duplicated for each thread and tagged following those states: *not accessed, modified, exposed loaded* or *exposed loaded and later modified*. For example, a *read-after-write* dependency has been violated if a thread owns a data tagged as *exposed loaded* or *exposed loaded and modified*, and if a predecessor thread, following the sequential total order, owns the same data but tagged as *modified* or *exposed loaded and modified*, while this data has not yet been committed in main memory. Such an approach can be memory-costly as each shared data structure is duplicated. It can be tricky to adjust verification frequencies to minimize time overhead. Some other methods based on the same principle of verifying speculation relatively to the sequential schedule have been proposed recently as in [65], where each iteration of a loop is decomposed into a prologue, a speculative body and an epilogue. The speculative bodies are performed in parallel and each body completion induces a verification. This approach seems to be only well suited for loops which bodies represent significant computation time.

Another recent work is the development of SPICE [60] which is a speculative parallelizing system where an entire first run of a loop is initially observed. This observation serves in determining the values reached by some variables during the run. During a next run of the loop, several speculative threads are launched. They consider as initial values of some variables the values that have been observed at the previous run. If a thread reaches the starting value of another thread, it stops. Thus each thread performs a different portion of the loop. But if the loop behavior changes and if another thread starting value is never reached, the run goes on sequentially until completion.

The main limits of these propositions are:

- they do not alter the initial sequential schedule since always contiguous instruction blocks are speculatively parallelized;
- their underlying parallelism is out of control: the characteristics of the generated parallel schedule are completely unknown since they randomly depend on the program instructions, their dependencies and the target machine. If bad performance is encountered, no other parallelization solution can be proposed. Moreover, the effective instruction schedule occurring at program run can significantly vary from one run to another, hence leading to a confusing performance inconsistency.

A strategy that would uniquely be based on a transactional memory mechanism, with rollbacks in the case of data races, yields a totally uncontrolable parallelism where performance can not be ensured and not even strongly expected.

While being based on efficient prediction mechanisms, a better control over parallelization will permit to provide solutions that are well suited to a varying execution context and to parallelize portions of code that can be parallelized only in some particular context. It is indeed crucial to maximize the potential parallelism of the applications to take advantage of the forthcoming processors comprising several tens of cores.

3.4.2. General objective: building a virtual machine

As it has already been mentioned, dynamic parallelization and optimization can take place inside a virtual machine. All the research objectives that are presented in the following are related to its construction.

Notice that the term of "virtual machine" is employed to group a set of dynamic analysis and optimization mechanisms taking as input a binary code, eventually enriched with specific instructions. We refer to a process virtual machine which main role is dynamic binary optimization from one instruction set to the same instruction set. The taxonomy given in [64] includes this kind of virtual machine.

Notice that this virtual machine can run in parallel on the processor cores during the four initial phases (see figure 2), but also simultaneously to the target application, either by sharing some cores with light processes, or by using cores that are useless for the target application. It will also support a transactional memory mechanism, if available. However the foreseen parallelizing strategies do not depend on such a mechanism since our speculative executions are supposed to be as reliable as possible thanks to efficient prediction models, and since they are supported by a specific and higher level rollback mechanism. Anyway if available, a transactional memory mechanism would allow to take advantage of "nearly perfect" prediction models.

The virtual machine takes as input an intermediate code expressing several kinds of parallelism on several code extracts. Those kinds of parallelism are either effective, that is to say that the corresponding parallel execution is obviously semantically correct, or hypothetical, that is to say that there is still some uncertainty on the parallelism correctness. In this case, this uncertainty will have to be resolved at run time. This intermediate “multi-parallel” code is generated by the static parallelization described subsection 3.2. It also contains generic descriptions of parallelizing or optimizing transformations which parameters will have to be instanciated by the virtual machine, thanks to its knowledge about the target architecture and the program run-time behavior.

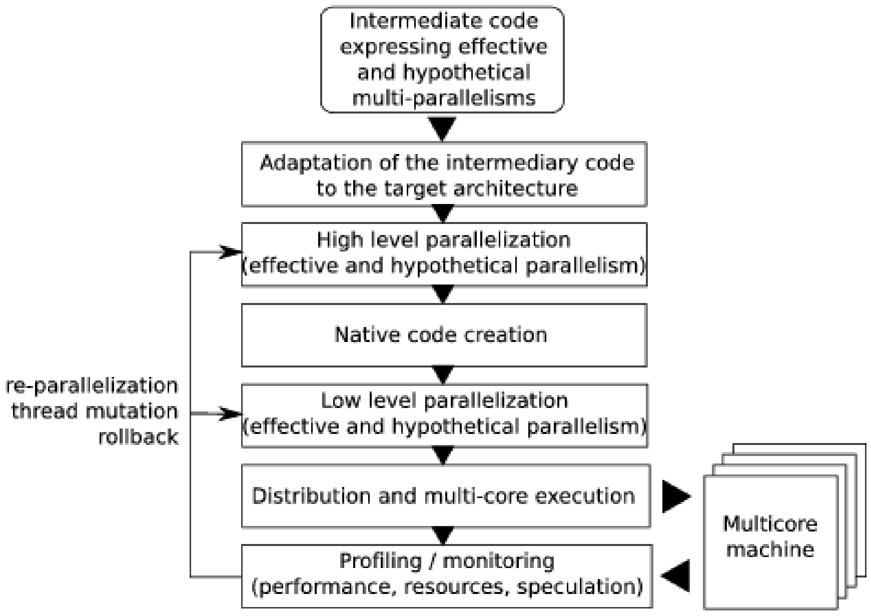
Intermediate code expressing effective and hypothetical multi-parallelisms
Adaptation of the intermediary code to the target architecture
High level parallelization (effective and hypothetical parallelism)
Native code creation
Low level parallelization (effective and hypothetical parallelism)
Distribution and multi-core execution
Profiling / monitoring (performance, resources, speculation)
re-parallelization thread mutation rollback
Multicore machine

Figure 2. The virtual machine

3.4.3. Adaptation of the intermediate code to the target architecture

The virtual machine first phase is to adapt this intermediate code to the target multicore architecture. It consists in answering the following questions:

- What is the suitable kind of parallelism?
- What is the suitable parallel task granularity?
- What is the suitable number of parallel tasks?
- Can we take advantage of a specialized instruction set for some operations?
- What are the parameter values for some parallelization or optimization?

The multi-parallel intermediate code exhibits different parameters allowing to adapt some parallelizing and optimizing transformations to the target architecture. For example, a loop unrolling will be parametrized by the number of iterations to be unrolled. This number will depend, for example, on the number of available registers and the size of the instruction cache. A parallelizing transformation will depend on several possible parallel instruction schedules. One or several schedules will be selected, for example, depending on the kind of memory hierarchy and the cache sharing among cores.

Concerning hypothetical parallelism, this first phase will reduce the number of these propositions to solutions that are well suited to the target architecture. This phase also instruments the intermediate code in order to install the dynamic mechanisms related to profiling and speculative parallel execution.

3.4.4. High level parallelization and native code creation

From these target architecture related adaptations, a parallel intermediate code is generated. It contains instructions that are specific to the dynamic optimizing and parallelizing mechanisms, *i.e.*, instrumentation instructions to feed the profiling process as well as calls to speculative execution management procedures. A translation into native code executable by the target processor follows. This translation also allows to keep trace of the code extracts that have to be modified during the run.

3.4.5. Low level parallelization

The binary version of the code exhibits new parallelism and optimization sources that are specific to the instruction set and to the target architecture capabilities. Moreover, some dynamic optimizations are dedicated to specific instructions, or instruction blocks, as for example the memory reads which time performances can be dynamically improved by data prefetching [1]. Thus the binary code can be transformed and instrumented as well.

3.4.6. Distribution, execution and profiling

The so built executable code is then distributed among the processor cores to be run. During the run, the instrumentation instructions feed the profiler with information for execution monitoring and for behavior models construction (see subsection 3.3). An accurate knowledge of the binary code, thanks to the control of its generation, also permits at this step to dynamically control the insertion or deletion of some instrumentation instructions. Indeed it is important to manage execution monitoring through sampling based instrumentations in varying frequencies, following the changing behavior frequency (see in [1] and [70] a description of this kind of mechanism), as such instrumentations necessarily induce overheads that have to be minimized.

3.4.7. Re-parallelization, thread mutation or rollback

Depending on the information collected from instrumentation, and depending on the built prediction models, the profiling phase causes a re-transformation of some code parts, thus causing the mutation of the concerned threads. Such re-transformation is done either on the binary code whether it consists in low level and small modifications, as for example the adjustement of a data prefetching distance, or on the intermediate code if it consists in a complete modification of the parallelizing strategy. For example, such a processing will follow the observation of a bad performance, or of a change in the computing resources availability, or will be caused by the completion of a dependency prediction model allowing the generation of a speculative parallelization. From such a speculative execution, a re-transformation can consist in rolling back to a sequential execution version when the considered hypothetical parallelism, and thus the associated prediction model, has been evaluated wrong.

3.5. Proof of program transformations for multicores

Participants: Éric Violard, Julien Narboux, Nicolas Magaud, Vincent Loechner, Alexandra Jimborean.

3.5.1. State of the art

3.5.1.1. Certification of low-level codes.

Among the languages allowing to exploit the power of multicore architectures, some of them supply the programmer a library of functions that corresponds more or less to the features of the target architecture : for example, CUDA[4] for the architectures of type GPGPU and more recently the standard OpenCL[5] that offers a unifying programming interface allowing the use of most of the existing multicore architectures or

[4] http://www.nvidia.com/object/cuda_what_is.html
[5] http://www.khronos.org/opencl

a use of heterogeneous aggregate of such architectures. The main advantage of OpenCL is that it allows the programmer to write a code that is portable on a large set of architectures (in the same spirit as the MPI library for multi-processor architectures). However, at this low level, the programming model is very close to the executing model, the control of parallelism is explicit. Proof of program correctness has to take into account low-level mechanisms such as hardware interruptions or thread preemption, which is difficult.

In [36], Feng *et al.* propose a logic inspired from the Hoare logic in order to certify such low-level programs with hardware interrupts and preempted threads. The authors specify this logic by using the meta-logic implemented in the Coq proof assistant [21].

3.5.1.2. Certification of a compiler.

The problem here is to prove that transformations or optimizations preserve the operational behaviour of the compiled programs.

Xavier Leroy in [24], [47] formalizes the analyses and optimizations performed by a C compiler: a big part of this compiler is written in the specification language of Coq and the executable (Caml) code of this compiler is obtained by automatic extraction from the specification.

Optimizing compilers are complex softwares, particularly in the case of multi-threaded programs. They apply some subtle code transformations. Therefore some errors in the compiler may occur and the compiler may produce incorrect executable codes. Work is to be done to remedy this problem. The technique of validation *a posteriori* [66], [67] is an interesting alternative to full verification of a compiler.

3.5.1.3. Semantics of directives.

As it was mentioned in subsection 3.2.3, the use of directives is an interesting approach to adapt languages to multicore architectures. It is a syntactic means to tackle the increasing need of enriching the operational semantics of programs.

Ideally, these directives are only comments: they do not alter the correction of programs and they are a good means to improve their performance. They allow the separation of concerns: *correction* and *efficiency*.

However, using directives in that sense and in the context of automatic parallelization, raises some questions: for example, assuming that directives are not mandatory, how to ensure that directives are really taken into account? How to know if a directive is better than another? What is the impact of a directive on performance?

In his thesis [38], that was supervised by Éric Violard, Philippe Gerner addresses similar questionings and states a formal framework in which the semantics of compilation directives can be defined. In this framework, any directive is encoded into one equation which is added to an algebraic specification. The semantics of the directives can be precisely defined via an order relation (called relation of *preference*) on the models of this specification.

3.5.1.4. Definition of a parallel programming model.

Classically, the good definition of a programming model is based on a semantic domain and on the definition of a “toy” language associated with a proof system, which allows to prove the correctness of the programs written in that language. Examples of such “toy” languages are CSP for control parallelism and $\mathcal{L}$ [26] for data parallelism. The proof systems associated with these two languages, are extensions of the Hoare logic.

We have done some significant works on the definition of data parallelism [11]. In particular, a crucial problem for the good definition of this programming model, is the semantics of the various syntactic constructs for data locality. We proposed a semantic domain which unifies two concepts: *alignment* (in a data-parallel language like HPF) and *shape* (in the data-parallel extensions of C).

We defined a “toy” language, called PEI, that is made of a small number of syntactic constructs. One of them, called *change of basis*, allows the programmer to exhibit parallelism in the same way as a placement or a scheduling directive [39].

3.5.1.5. Programming models for multicore architectures.

The multicore emergence questions the existing parallel programming models.

For example, with the programming model supported by OpenMP, it is difficult to master both correctness and efficiency of programs. Indeed, this model does not allow programmers to take optimal advantage of the memory hierarchy and some OpenMP directives may induce unpredictable performances or incorrect results.

Nowadays, some new programming models are experienced to help at designing both efficient and correct programs for multicores. Because memory is shared by the cores and its hierarchy has some distributed parts, some works aim at defining a hybrid model, between task parallelism and data parallelism. For example, languages like UPC (Unified Parallel C)[6] or Chapel[7] combine the advantages of several programming paradigms.

In particular, the model of memory transactions (or transactional memory [44]) retains much attention since it offers the programmer a simple operational semantics including a mutual exclusion mechanism which simplifies program design. However, much work remains to define the precise operational meaning of transactions and the interaction with the other languages features [53]. Moreover, this model leaves the compiler a lot of work to reach a safe and efficient execution on the target architecture. In particular, it is necessary to control the atomicity of transactions [37] and to prove that code transformations preserve the operational semantics.

3.5.1.6. Refinement of programs.

Refinement [19], [40] is a classical approach for gradually building correct programs: it consists in transforming an initial specification by successive steps, by verifying that each transformation preserves the correctness of the previous specification. Its basic principle is to derive simultaneously a program and its own proof. It defines a formal framework in which some rules and strategies can be elaborated to transform specifications written by using the same formalism. Such a set of rules is called a *refinement calculus*.

Unity [30] and Gamma [20] are classical examples of such formalisms, but they are not especially designed for refining programs for multicore architectures. Each of these formalisms is associated with a computing model and thus each specification can be viewed as a program. Starting with an initial specification, a proof logic allows a user to derive a specification which is more suited to the target architecture.

Refinement applies for the programming of a large range of problems and architectures. It allows to pass the limitations of the polyhedral model and of automatic parallelization. We designed a refinement calculus to build data parallel programs [68].

3.5.2. Main objective: formal proof of analyses and transformations

Our main objective consists in certifying the critical modules of our optimization tools (the compiler and the virtual machine). First we will prove the main loop transformation algorithms which constitute the core of our system.

The optimization process can be separated into two stages: the transformations consisting in optimizing the sequential code and in exhibiting parallelism, and those consisting in optimizing the parallel code itself. The first category of optimizations can be proved within a sequential semantics. For the other optimizations, we need to work within a concurrent semantics. We expect the first stage of optimizations to produce data-race free code. For the second stage of optimizations, we will first assume that the input code is data-race free. We will prove those transformations using Appel's concurrent separation logic [42]. Proving transformations involving program which are not data-race free will constitute a longer term research goal.

3.5.3. Proof of transformations in the polyhedral model

The main code transformations used in the compiler and the virtual machine are those carried out in the polyhedral model [46], [35]. We will use the Coq proof assistant to formalize proofs of analyses and transformations based on the polyhedral model. In [29], Cachera and Pichardie formalized nested loops in Coq and showed how to prove *properties* of those loops. Our aim is slightly different as we plan to prove *transformations* of nested loops in the polyhedral model. We will first prove the simplest unimodular

[6]http://upc.gwu.edu
[7]http://chapel.cs.washington.edu

transformations, and later we will focus on more complex transformations which are specific to multicore architectures. We will first study scheduling optimizations and then optimizations improving data locality.

3.5.4. Validation under hypothesis

In order to prove the correction of a code transformation T it is possible to:

- prove that T is correct in general, *i.e.*, prove that for all x, $T(x)$ is equivalent to x.
- prove *a posteriori* that the applied transformation has been correct in the particular case of a code c.

The second approach relies on the definition of a program called *validator* which verifies if two pieces of program are equivalent. This program can be modeled as a function V such that, given two programs c_1 and c_2, $V(c_1, c_2) = true$ only if c_1 has the same semantics as c_2. This approach has been used in the field of optimizations certification [56], [55]. If the validator itself contains a bug then the certification process is broken. But if the validator is proved formally (as it was achieved by Tristan and Leroy for the Compcert compiler [66], [67]) then we get a transformed program which can be trusted in the same way as if the transformation is proved formally.

This second approach can be used only for the *effective parallelism*, when the static analysis provides enough information to parallelize the code. For the *hypothetical parallelism*, the necessary hypotheses have to be verified at run time.

For instance, the absence of aliases in a piece of code is difficult to decide statically but can be more easily decided at run time.

In this framework, we plan to build a *validator under hypotheses*: a function V' such that, given two programs c_1 and c_2 and an hypothesis H, if $V'(c_1, c_2, H) = true$, then H implies that c_1 has the same semantics as c_2. The validity of the hypothesis H will be verified dynamically by the virtual machine. This verification process, which is part of the virtual machine, will have to be proved as correct as well.

3.5.5. Rejecting incorrect parallelizations

The goal of the project is to exhibit potential parallelism. The source code can contain many sub-routines which could be parallelized under some hypothesis that the static analysis fails to decide. For those optimizations, the virtual machine will have to verify the hypotheses dynamically. Dynamically dealing with the potential parallelism can be complex and costly (profiling, speculative execution with rollbacks). To reduce the overhead of the virtual machine, we will have to provide efficient methods to rule out quickly incorrect parallelism. In this context, we will provide hypotheses which are easy to check dynamically and which can tell when a transformation cannot be applied, *i.e.*, hypotheses which are sufficient conditions for the non-validity of an optimization.

4. Application Domains

4.1. Application Domains

Performance being our main objective, our developments' target applications are characterized by intensive computation phases. Such applications are numerous in the domains of scientific computations, optimization, data mining and multimedia.

Applications involving intensive computations are necessarily high energy consumers. However this consumption can be significantly reduced thanks to optimization and parallelization. Although this issue is not our prior objective, we can expect some positive effects for the following reasons:

- Program parallelization tries to distribute the workload equally among the cores. Thus an equivalent performance, or even a better performance, to a sequential higher frequency execution on one single core, can be obtained.

- Memory and memory accesses are high energy consumers. Lowering the memory consumption, lowering the number of memory accesses and maximizing the number of accesses in the low levels of the memory hierarchy (registers, cache memories) have a positive consequence on execution speed, but also on energy consumption.

5. Software

5.1. PolyLib

PolyLib[8] is a C library of polyhedral functions, that can manipulate unions of rational polyhedra of any dimension, through the following operations: intersection, difference, union, convex hull, simplify, image and preimage. It was the first to provide an implementation of the computation of parametric vertices of a parametric polyhedron, and the computation of an Ehrhart polynomial (expressing the number of integer points contained in a parametric polytope) based on an interpolation method.

It is used by an important community of researchers (in France and the rest of the world) in the area of compilation and optimization using the polyhedral model. Vincent Loechner is the maintainer of this software. It is distributed under GNU General Public License version 3 or later, and it has a Debian package maintained by Serge Guelton (Symbiose Projet, IRISA).

5.2. NLR

We have developed a program implementing our loop-nest recognition algorithm, detailed in [7]. This standalone, filter-like application takes as input a raw trace and builds a sequence of loop nests that, when executed, reproduce the trace. It is also able to predict forthcoming values at an arbitrary distance in the future. Its simple, text-based input format makes it applicable to all kinds of data. These data can take the form of simple numeric values, or have more elaborate structure, and can include symbols. The program is written is standard ANSI C. The code can also be used as a library.

We have used this code to evaluate the compression potential of loop nest recognition on memory address traces, with very good results. We have also shown that the predictive power of our model is competitive with other models on average. The software is available upon request to anybody interested in trying to apply loop nest recognition. It has been distributed to a dozen of colleagues around the world.

We plan on using this software as the base for a new tool we currently design, for the analysis of parallel traces.

5.3. Dynamic version selector

We are developing a toolchain to automatically select between different versions of parallel loop nests, as described in subsection 6.1. It generates the profiling code and selection code from a loop nest source code and different schedules, expressed in the CLooG format.

Benoit Pradelle (PhD) wrote this toolchain, based on python scripts. It is not yet distributed.

5.4. Binary files decompiler

Our research on efficient memory profiling has lead us to develop a sophisticated decompiler. This tool analyzes x86-64 binary programs and libraries, and extracts various structured representations of the code. It works on a routine per routine basis, and first builds a loop hierarchy to characterize the overall structure of the algorithm. It then puts the code into Static Single Assignment (SSA) form to highlight the fine-grain data-flow between registers and memory. Building on these, it performs the following analyzes:

[8] http://icps.u-strasbg.fr/PolyLib

- All memory addresses are expressed as symbolic expressions involving specific versions of register contents, as well as loop counters. Loop counter definitions are recovered by resolving linearly incremented registers and memory cells, i.e., registers that act as induction variables.
- Most conditional branches are also expressed symbolically (with registers, memory contents, and loop counters). This captures the control-flow of the program, but also helps in defining what amounts to loop "trip-counts", even though our model is slightly more general, because it can represent any kind of iterative structure.

This tool embodies several passes that, as far as we know, do not exist in any existing similar tool. For instance, it is able to track data-flow through stack slots in most cases. It has been specially designed to extract a representation that can be useful in looking for parallel (or parallelizable) loops [13]. It is the basis of several of our studies.

Because binary program decompilation is especially useful to reduce the cost of memory profiling, our current implementation is based on the Pin binary instrumenter. It uses Pin's API to analyze binary code, and directly interfaces with the upper layers we have developed (e.g., program skeletonization, or minimal profiling). However, we have been careful to clearly decouple the various layers, and to not use any specific mechanism in designing the binary analysis component. Therefore, we believe that it could be ported with minimal effort, by using a binary file format extractor and a suitable binary code parser. It is also designed to abstract away the detailed instruction set, and should be easy to port (even though we have no practical experience in doing so).

We feel that such a tool could be useful to other researchers, because it makes binary code available under abstractions that have been traditionally available for source code only. If sufficient interest emerges, e.g., from the embedded systems community, or from researchers working on WCET, or from teams working on software security, we are willing to distribute and/or to help make it available under other environments.

5.5. Dynamic depency analyser

We have recently started developing a dynamic dependence analyzer. Such a tool consumes the trace of memory (or object) accesses, and uses the program structure to list all the data dependences appearing during execution. Data dependences in turn are central to the search for parallel sections of code, with the search for parallel loops being only a particular case of the general problem. Most current works of these questions are either specific to a particular analysis (e.g., computing dependence densities to select code portions for thread-level speculation), or restricted to particular forms of parallelism (e.g., typically to fully parallel loops). Our tool tries to generalize existing approaches, and focuses on the program structures to provide helpful feedback either to a user (as some kind of "smart profiler"), or to a compiler (for feedback-directed compilation). For example, the tool is able to produce a dependence schema for a complete loop nest (instead of just a loop). It also targets irregular parallelism, for example analyzing a loop execution to estimate the expected gain of parallelization strategies like inspector-executor.

We have developed this tool in relation to our minimal profiling research project. However, the tool itself has been kept independent of our profiling infrastructure, getting data from it via a well-defined trace format. This intentional design decision has been motivated by our work on distinct execution environments: first on our usual x86-64 benchmark programs, and second on less regular, more often written in Java, real-world applications. The latter type of applications is likely the one that will most benefit from such tools, because their intrinsic execution environment does not offer enough structure to allow effective static analysis techniques. Parallelization efforts in this context will most likely rely on code annotations, or specific programming language constructs. Programmers will therefore need tools to help them choose between various constructs. Our tool has this ambition. We already have a working tool-chain for C/C++/Fortran programs (or any binary program). We are in the process of developing the necessary infrastructure to connect the dynamic dependence profiler to instrumented Java programs. Other managed execution environments could be targeted as well, e.g., Microsoft's .Net architecture, but we have no time and/or workforce to devote to such time-consuming engineering efforts.

5.6. VMAD software and LLVM

For dynamic analysis and optimization of programs, we developed a virtual machine called VMAD, and specific passes to the LLVM compiler suite, plus a modified Clang frontend. It is fully described in subsection 6.2.

We implemented for now a memory access predictor in loop nests, based on the computation of linear interpolation functions. The profiling is very fast compared to other existing tools, as it samples only the first few iterations of each loop in the nest, then it is deactivated to return to the original, faster version. Other tools like PIN or PEBIL do not support such activation/deactivation mechanism.

New annotations for the final user, taken as input by LLVM, and new VMAD modules will be developed, as these tools have been designed to be very evolving.

Alexandra Jimborean (PhD) and Matthieu Herrmann (Master student) wrote this software. It is not yet distributed.

5.7. Polyhedral prover

Participants: Nicolas Magaud, Julien Narboux, Éric Violard [correspondant].

We are currently developing a formal proof of program transformations based on the polyhedral model. We use the CompCert verified compiler [48] as a framework. This tool is written in the specification language of Coq. It is connected to the activity described in section 6.7.

6. New Results

6.1. Dynamic version selector

Adaptive version selection between different parallel versions of code is necessary when the execution context of a program is not known. The execution contexts includes all or some of these possibly variable parameters: the target architecture, the load of the computer at execution time, and the input data.

We have developed a framework handling loops in the polyhedral model, that is able to take a runtime decision about which version to execute. It is based on :

- the generation of different code versions of a loop nest;
- an install-time profiling to take into account the architecture parameters, that builds a parametric ranking table between the versions;
- a runtime selection, predicting the load balance and the execution time of each code version, before executing the best one.

We showed that different versions of a code are required on several polyhedral loop nest benchmarks, depending on both the target architecture and the input data. And we showed speedups compared to any statically chosen version in all execution contexts. More details are available in the research report [17].

6.2. VMAD and LLVM

The goal is to provide a set of annotations (pragmas) that the user can insert in the source code to perform low level analyses (profiling) or optimizations (dynamic parallelisation for example).

We started the development of a virtual machine and an efficient implementation of advanced profiling and analysis of programs. VMAD is organized as a sequence of basic operations, where external modules associated to specific strategies are dynamically loaded when required. The program binary files handled by VMAD are previously instrumented at compile time to include necessary data, instrumentation instructions and callbacks to the virtual machine. Dynamic information, such as memory locations of launched modules, are patched at startup in the binary file. The LLVM compiler has been extended to automatically instrument programs to meet the requirements both of VMAD and of the handled/chosen profiling strategies.

A profiling strategy interpolating the memory addresses accessed in a loop nest has been run on some of the SPEC2006 and Pointer Intensive benchmark suites, showing a very low time overhead, in most cases. More details are available in the research reports [16] and [15], and publication [12].

We are now working on the development of other profiling and optimization techniques, and on a generic API that could be used by the experienced programmer to write his own instrumentations.

6.3. Static analysis of the memory behavior of executable programs

For the last year and a half, we have been developing techniques to analyze binary code. The major goal is to find out, from a binary executable program and its libraries, how executing this program will use memory. When the program's memory behavior is characterized in some abstract (and parametrized) way, several studies can be conducted directly on the binary program, without any need of the source code, which may be totally or partially unavailable. Our own research topics have several examples of interesting facts that can be derived directly. The first example is that of "program skeletonization", a lightweight instrumentation strategy to obtain a full memory trace (e.g., for dynamic dependence analysis, or cache simulation). Another example is that of the static parallelization of binary code, i.e., building a parallelizing compiler that works directly on executable programs and can thus handle mixed language programs, programs build with proprietary libraries, and so on. In all cases, it appeared that obtaining the memory behavior of the binary program was a central element. Existing systems were rudimentary, and the various techniques and algorithms that we had to use or develop specifically are significant enough that we have decided to promote this topic as a scientific result.

Our approach is to consider a binary program with the program structures that are commonly used in compiler techniques. The code is split into basic blocks organized into a control-flow graph, and this control-flow graph is structured as a hierarchy of loops. This can be done with well-known, textbook techniques, and many systems have used a similar approach (even though our experience suggests many of them fail to deal properly with some binary code specific artifacts, like irreducible loops). From that point, approaches vary widely, but no systematic and efficient technique seemed precise and accurate enough. We have chosen to base the rest of the analysis on the Static Single Assignment form, which has been tremendously useful in developing source level or intermediate representation level compiler optimizations. We have shown that, by not trying to be overly precise from early on, one can extract an interesting representation of a binary program from its binary code.

Extracting the memory behavior of a program means constructing a representation of the program that explicitly expresses how the program will access memory, in a manner that is amenable to detailed analysis. The hierarchy of loops provides the global structure, and an SSA-based symbolic analysis details how the individual memory accesses vary from one iteration to the other. This requires two analysis phases after the program is in SSA form. The first phase is mixing program slicing and forward substitution to express every memory address symbolically. The second phase focuses on loops and determines which registers hold values that vary linearly across loop iterations. The resulting representation is a program where memory accesses are defined by linear combinations of loop counters and parameters (specific register versions), with the latter hopefully loop invariant. This representation is actually very similar to the one used for static control programs (when applicable), and forms the basis of almost all parallelization techniques. These results have already been published [13], and the conference program committee has invited us to prepare an extended version of this paper for publication in an international journal in 2011.

Building on this foundation, we have developed several techniques to enhance our basic strategy. The major limitation of our strategy, as just described, is that it restricts itself to address computations that happen completely inside the registers, ignoring any flow of data to and from memory. Fully characterizing this traffic being clearly out of reach (it would mean completely solving a general parametrized dependence problem), we have chosen to solve a restricted problem: separating access to the current stack-frame from the accesses to the rest of the memory. This requires a specific form of points-to analysis, that is able to determine whether a given address "points to" a location inside the current stack-frame, or to a location outside of it. In many cases this lets the system determine that two memory address expressions either *cannot* alias, because they point to separate portions of memory, or may alias, in which case a simple and conservative comparison of address expressions can decide whether they designate the same memory location. We solve this problem with a forward data-flow analysis. It appears that most of the time, this approach is enough to let the system track the flow of data through the stack-slots, which, in turn, provides some rough equivalent to use-def links for these stack slots. When used to derive symbolic expressions for memory addresses (as explained above), it lets the slicing process go further back, and lets the induction variable resolution apply to stack slots. The overall result is that more loops are completely understood by the decompilation process.

6.4. Dynamic analysis of the memory behavior of executable programs

Program skeletonization consists in transforming a given program into another program whose only task is to produce some data that one wants to observe about the original program. We apply this technique to memory tracing. If one wants to obtain the complete list of memory address that the given program accesses, our algorithm builds a new program that outputs the list of memory addresses. This new program, called the *skeleton*, is not equivalent to the original program: it is restricted to what the user is interested in (in our case, memory addresses). The rest of the program, e.g., the computation of results, is simply ignored and does not appear in the skeleton. The main motivation for skeletonization is practical: instrumenting each and every memory access in a given program has two main difficulties. First the original program increases in size, which makes it slower. Second, since memory accesses are extremely frequent (on every third instruction execution on average), the instrumentation usually causes massive slowdowns.

Because we want to reproduce the list of memory addresses *for a given execution*, the skeleton's execution needs to somehow depend on the input data. Program skeletonization is designed to clearly separate both aspects: the skeleton is directly derived from the original program only, but it needs an input trace to reproduce a given execution. However, this input trace may not be composed exclusively of relevant input data. It may also contain intermediate computations that have been found too complex for the skeleton to reproduce, and these intermediate computations may or may not use input data. The only guarantees provided are: 1) the skeleton is completely independent of input data, and 2) given its input trace, the skeleton will faithfully reproduce the stream of memory access addresses.

Building the skeleton is an immediate application of our decompilation process (described earlier). Starting with a binary program, the decompiler provides 1) a linear combination of loop counters and base registers for each memory access, 2) a simple comparison of a linear form for all branch conditions that can be parsed, and 3) a set of base register definitions, that are used in the various expressions. Building the skeleton is immediate: our algorithm generates one basic block per basic block in the original program, where it places input statements where a base register is defined, output statements where an address is computed, and branches if the basic block ends with a conditional branch (the condition governing this branch is either computed or input). Loop counters are also defined and incremented in the skeleton, but this does not require any input data. Our system actually generates a C program, giving the C compiler an additional opportunity to optimize the skeleton code.

Producing the skeleton's input trace is the charge of the original program, which must be instrumented to output the values of the base registers. This is a regular instrumentation, obtaining register values, as well as unknown branch outcomes. Every obtained value is written out, either to a pipe at the end of which the skeleton is currently running, or to a file if the trace has to be saved to be reused later (possibly multiple times).

The overall strategy is interesting in several respects. First, the skeleton is independent of the input data, which means it is reusable across executions. Second, the input trace produced by instrumenting the original program is usually much smaller than the full memory trace, which makes the original program run with less overhead. This is true statically (there are less instrumentation points than for memory instrumentation) and dynamically (since many loops are completely based on loop invariant registers, a large part of the execution doesn't use instrumentation code at all). Finally, running the skeleton is completely independent of the original execution context, which is not needed anymore since every aspect of it has been captured in the trace.

A paper describing our approach and system has been accepted for presentation at the *2011 IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS-2011) [14], to be held in April.

6.5. Dynamic dependency analysis

Dynamic dependence analysis consists in observing all memory (or object) accesses performed by a program to detect all the data dependences that occur. It is essentially incomplete in that it relies on a specific execution, but can provide insight on the memory behavior and help designing correct program transformations, e.g., for parallelization. Dynamic dependence analysis has been used mainly for detecting parallel, or almost parallel, program sections, and are often coupled with thread-level speculation. We believe dynamic dependence analysis can also be extremely useful as an aid to the programmer, provided the tools are able to extract usable information on which parts of the program can be parallelized. We plan to make it a central component to our parallelization assistant project.

In its simplest form, dynamic dependence analysis takes a trace of memory accesses, and finds in the trace the addresses that have appeared previously. Every such occurrence is a dependence if any of the accesses involved is a write. This is barely useful if it is not related in some way to the structure of the program under scrutiny. Most previous studies have used “local” structures, keeping, for instance, a table of memory accesses for each iteration of a loop, and then testing independence to find out that the loop is parallel (in the sense that all its iterations could have been run in parallel). Even when the loop iterations are dependent, a low frequency of dependence may still make the loop be considered a good candidate for speculative parallelization.

Our approach is much more focused on the program structure. It associates to each memory access an execution point that contains an interleaving of function calls and loop iterations (we call this an extended stack trace). Given two conflicting accesses, the longest common prefix of their extended stack trace indicates the “nearest” program structure that exhibits the dependence (either a function call or a loop iteration). This has been used in other works, to detect parallel loops. What is less often noticed is that the remaining portions of the execution points (i.e., the upper parts of the stacks) provide detailed information on the respective iterations that carry the dependence. It is therefore possible to extract complete dependence schema for loop nests (with dependence distances), and not only for single loops as is usual. This means that, if dynamic dependence analysis is used in an interactive tool, more elaborate parallelization strategies can be suggested.

Another major topic related to dynamic dependence analysis is the scalability of the approach, which requires heavy instrumentation and sophisticated and efficient data structures to be usable. Part of the solution is covered by our other studies on “program skeletonization” (see above), but much remains to be done to make the analysis usable in a developer tool. We have started work on using static analysis of the binary program (and/or debugging information where available) to reduce the cost of instrumentation and analysis. Once again, this relies on detecting loops whose memory footprints can be known by analyzing the code. Fortunately, a large body of theory, especially on parametric integer programming, can help reduce the amount of data to process. The idea is to use the static description of complete parts of the program to update the dependence data structures. This is again an illustration of a principle that is used throughout our work, namely the cooperation between static and dynamic data.

6.6. Binary parallelization

Our basic decompilation mechanism lets us extract loop nests from binary code. In favorable cases, the loops are fully described, and every memory access they contain has an associated linear combination of base register values and loop counters. Such a representation is enough to apply parallelization techniques based on the polytope model. These techniques are currently the best known way to derive an efficient parallel equivalent of the original loop nest. Our goal in this project is to exploit this similarity of representations, to build a parallelizing binary-to-binary compiler. Given that we are able to locate and describe loop nests inside binary programs, the basic workflow of our compiler is to first extract a suitable representation of the complete loop nest, then, in a second phase, apply parallelizing transformations to the model, and then, in the third and last phase, regenerate binary code for the parallel version of the loop nest. The rest of the program remains unchanged.

The first phase, called the *raising* phase, relies on our decompilation techniques. Given a binary program, the loops are located and brought (or "raised") to a data structure containing linear memory access functions and loop bounds. All functions may reference loop invariant registers, whose definitions are also precisely located, thanks to the SSA form. The first step is to remove unnecessary instructions: since all memory accesses are expressed as functions, the parts of the program that participate in actually computing these functions becomes useless. We have employed a basic slicing technique to select the instructions that need to be preserved. Starting from instructions that write either to memory or a register that is live on exit of the loop, our algorithm follows all possible paths through definitions that are required for the written values to be computed. Redundant instructions (i.e., those that have not been visited during the slicing) are removed. The remaining instructions are all necessary for the transformed program to have the same effect as the original program. These instructions, as well as the loop nest structure, are translated into C-like code. Various other transformations are also applied in order to "clean up" the extracted program and make it usable by the next phase.

The second phase, called the parallelization *phase*, takes as input our C-like loop nest filled with affine accesses to a unique array (the memory). It then performs a complete parallelization analysis, including dependence analysis and scheduling. We have left this part to an external tool, not developed by our team [25]. This tool is a state-of-the-art parallelizer, and provides the basic infrastructure. However, its use is, from our point of view, a temporary solution: it will be complemented by various other tools in the future. The result of this phase is a new program, augmented with compiler directives (typically, OpenMP directives).

The third and last phase, called the *lowering* phase, consists in reproducing executable code. As we have seen, most of the work happens during the first phase (because parallelization is performed by a "real" compiler). However, conceptually, it is relevant to consider it a distinct phase. It mostly consists in ensuring that all groups of instructions that have an effect are actually present in the result. It mostly involves translating low-level instructions extracted from the binary code into C code that are guaranteed to be propagated almost directly into the final program. It also involves creating a new executable including the modified code and ensure the correct integration inside the code that is copied verbatim. This phase is mostly technical, even though it has led us to develop several techniques that may be reused in other contexts.

This project is especially interesting because it provides a new view on parallelization (very few attempts have been made at general-purpose, binary-to-binary compilers). Its main contribution is to make parallelization a service of the execution environment, instead of a feature of the compiler. Any user of the operating system can benefit from our parallelizer, whatever compiler they used to produce the original sequential programs, and whatever library they used, even for programs that use components written in different languages and/or compiled with different compilers. Deferring the parallelization to the execution environment opens up several new research directions, which we plan to explore as soon as possible. It also allows the parallelization to take into account the target architecture.

6.7. Proof of polyhedral transformations

Participants: Nicolas Magaud, Julien Narboux, Éric Violard.

We work with two members of the GALLIUM team: Alexandre Pilkiewicz, PhD student, and François Pottier, senior researcher at INRIA. This work aims at integrating the polyhedral transformations into the compiler CompCert.

This integration is based on an *ad hoc* language called LOOPS. This language, designed by Alexandre Pilkiewicz, is a small abstract language (without concrete syntax) allowing to express the *affine loop nests* to which the polyhedral transformations apply. It is provided with a small-step operational semantics.

We use this language to separate proofs about polyhedral transformations from the actual intermediate languages of the compiler hence some proofs can be developped independently of CompCert.

The integration in CompCert takes place at the level of the intermediate language Csharpminor (cf. fig 3): the affine loop nests are first extracted from the intermediate code Csharpminor and translated into LOOPS. Once transformed while preserving their semantics, these loop nests are then translated back into Csharpminor.

We developed an extension of CompCert who extracts affine loop nests and performs the operation of transplantation. This extension is still rudimentary: only a restricted class of affine loop nests is recognized and translated into LOOPS.

We now work to establish the proof in Coq that, if the transformation in LOOPS is correct, then the transformed Csharpminor program has the same behavior as the original one.

We will have to deal with the problem of potential overflows in loop bounds computations. This problem is overlooked in the litterature about polyhedral model. We will have to generate some sufficient conditions to prevent overflow and check at runtime these conditions to guarrantee that the new program has the same behaviour as the original.

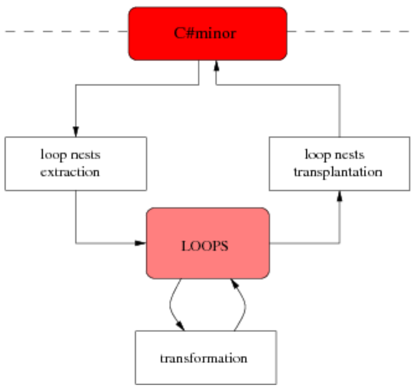

Figure 3. Integration within CompCert

7. Other Grants and Activities

7.1. National Initiatives

Philippe Clauss, Alain Ketterlin and Vincent Loechner are involved in the proposition of an INRIA Large Wingspan Project (*Action d'Envergure Nationale*) entitled "Software for multicores and hardware accelerators" and regrouping several french teams doing researches in compilers, parallel computing and program optimization.

7.2. International Initiatives

Since 2005, there has been a strong partnership between CAMUS and the team LAFHIS from the University of Buenos Aires, Argentina. The persons involved in this collaboration are: Philippe Clauss, Vincent Loechner and Alain Ketterlin from the CAMUS team; Sergio Yovine and Diego Garbervetsky from the LAFHIS team [9]. Martin Rouaux, who is an argentinian student, has began his PhD work in September 2010. This thesis is co-supervised by Diego Garbervetsky and Philippe Clauss. Martin Rouaux has spent four months in the CAMUS team between August and November 2010. His PhD subject is to develop a framework dedicated to analyzing precisely the object accesses made in object-oriented programs, in order to identify memory dependences and parallelization opportunities. The investigated approach is based on the improvement of static analysis techniques thanks to the support of dynamic analyses.

[9] http://www.lafhis.dc.uba.ar

Memory behavior analysis of object-oriented programs is the common issue of both teams. From 2007 to 2009, in our joint project from ECOS-Sud "Japiqay", we developped an original approach to evaluate statically the maximum amount of memory consumed by a java program using dynamic memory allocation [4].

This year, we submitted an international cooperation project MINCYT/INRIA/CNRS "QUATRIX" that was accepted. It will allow us to do some PhD students exchanges in the next months.

8. Dissemination

8.1. Animation of the scientific community

Julien Narboux and Nicolas Magaud organized a one day meeting of the french working group Language, Types, and Proofs of the GDR GPL[10].

Julien Narboux participated in the program committee of ADG 2010[11] and SCGD 2011[12].

Philippe Clauss participates to the program committee of the *First International Workshop on Polyhedral Compilation Techniques, IMPACT 2011*[13], that will be held in April 2011.

Philippe Clauss participated to the following jurys in 2010:

Date	**Thesis**	**Candidate**	**Place**	**Role**
Sept. 3	PhD	C. Ballabriga	Univ. Toulouse	Rewiever
Oct. 29	PhD	Q. Meunier	Univ. Grenoble	President
Déc. 9	PhD	D. Hardy	Univ. Rennes I	Examiner
Déc. 15	HDR	N. Wicker	Univ. Strasbourg	"Garant" & Examiner

The PhD works taking place currently in the CAMUS team are:

- Started December 2008: Benoît Pradelle, *Dynamic parallelization for multicore architectures*;
- Started October 2009: Alexandra Jimborean, *Advanced parallelization tools for multicore programming*;
- Started October 2010: Martìn Rouaux, *Memory behavior capture for the parallelization of object-oriented programs*, in co-supervision with the University of Buenos Aires and Prof. Diego Garbervetsky.

8.2. Teaching

As Professor and Assistant Professors at the University of Strasbourg, Philippe Clauss, Nicolas Magaud, Julien Narboux and Éric Violard each gave more than 200 hours of lectures. Benoît Pradelle, as Assistant teacher (*moniteur*), also gave about 64 hours of lectures.

Julien Narboux gave a lecture about Interactive Theorem Proving at master 1 level and a lecture about Certification of Programs at master 2 level.

9. Bibliography

Major publications by the team in recent years

[1] J. C. Beyler, P. Clauss. *Performance driven data cache prefetching in a dynamic software optimization system*, in "ICS '07: Proceedings of the 21st annual international conference on Supercomputing", New York, NY, USA, ACM, 2007, p. 202–209, http://doi.acm.org/10.1145/1274971.1275000.

[10] https://lsiit-cnrs.unistra.fr/gdr-ltp-2010/index.php/Accueil
[11] https://lsiit-cnrs.unistra.fr/adg2010/index.php/Main_Page
[12] http://webs.uvigo.es/fbotana/scdg2011
[13] http://impact2011.inrialpes.fr

[2] J. C. BEYLER, M. KLEMM, P. CLAUSS, M. PHILIPPSEN. *A meta-predictor framework for prefetching in object-based DSMs*, in "Concurr. Comput. : Pract. Exper.", September 2009, vol. 21, p. 1789–1803, http://dx.doi.org/10.1002/cpe.v21:14.

[3] P. CLAUSS. *Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs*, in "ICS '96: Proceedings of the 10th international conference on Supercomputing", New York, NY, USA, ACM, 1996, p. 278–285, http://doi.acm.org/10.1145/237578.237617.

[4] P. CLAUSS, F. J. FERNÁNDEZ, D. GARBERVETSKY, S. VERDOOLAEGE. *Symbolic polynomial maximization over convex sets and its application to memory requirement estimation*, in "IEEE Transactions on Very Large Scale Integration (VLSI) Systems", Aug 2009, vol. 17, n^{o} 8, p. 983-996, http://hal.inria.fr/inria-00504617.

[5] P. CLAUSS, V. LOECHNER. *Parametric Analysis of Polyhedral Iteration Spaces*, in "J. VLSI Signal Process. Syst.", 1998, vol. 19, n^{o} 2, p. 179–194, http://dx.doi.org/10.1023/A:1008069920230.

[6] P. CLAUSS, I. TCHOUPAEVA. *A Symbolic Approach to Bernstein Expansion for Program Analysis and Optimization*, LNCS, Springer, April 2004, vol. 2985, p. 120-133.

[7] A. KETTERLIN, P. CLAUSS. *Prediction and trace compression of data access addresses through nested loop recognition*, in "6th annual IEEE/ACM international symposium on Code generation and optimization", États-Unis Boston, ACM, April 2008, p. 94-103, http://dx.doi.org/10.1145/1356058.1356071, http://hal.inria.fr/inria-00504597/en.

[8] V. LOECHNER, B. MEISTER, P. CLAUSS. *Precise data locality optimization of nested loops*, in "Journal of Supercomputing", January 2002, vol. 21, n^{o} 1, p. 37–76, Kluwer Academic Pub..

[9] V. LOECHNER, D. K. WILDE. *Parameterized Polyhedra and their Vertices*, in "International Journal of Parallel Programming", December 1997, vol. 25, n^{o} 6.

[10] S. VERDOOLAEGE, R. SEGHIR, K. BEYLS, V. LOECHNER, M. BRUYNOOGHE. *Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions*, in "Algorithmica", 2007, vol. 48, n^{o} 1, p. 37–66, http://dx.doi.org/10.1007/s00453-006-1231-0.

[11] É. VIOLARD. *A Semantic Framework to Address Data Locality in Data Parallel Languages*, in "Parallel Computing", 2004, vol. 30, n^{o} 1, p. 139-161.

Publications of the year

International Peer-Reviewed Conference/Proceedings

[12] A. JIMBOREAN, M. HERRMANN, V. LOECHNER, P. CLAUSS. *VMAD: a Virtual Machine for Advanced Dynamic Analysis of Programs*, in "International Symposium on Performance Analysis of Systems and Software, ISPASS", États-Unis Austin, IEEE (editor), Apr 2011, http://hal.inria.fr/inria-00544501/en.

[13] A. KETTERLIN, P. CLAUSS. *Recovering the Memory Behavior of Executable Programs*, in "10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM", Roumanie Timisoara, IEEE Computer Society Press, Sep 2010, http://hal.inria.fr/inria-00502813.

[14] A. KETTERLIN, P. CLAUSS. *Efficient Memory Tracing by Program Skeletonization*, in "International Symposium on Performance Analysis of Systems and Software, ISPASS", États-Unis Austin, IEEE (editor), Apr 2011, http://hal.inria.fr/inria-00544497/en.

Research Reports

[15] A. JIMBOREAN, M. HERRMANN, V. LOECHNER, P. CLAUSS. *A Static-Dynamic Collaborative Framework for Nested Loops Instrumentation and Profiling*, Université de Strasbourg, 05 2010, http://hal.inria.fr/inria-00534745/en/.

[16] A. JIMBOREAN, M. HERRMANN, V. LOECHNER, P. CLAUSS. *VMAD: a Virtual Machine for Advanced Dynamic Analysis of Programs*, Université de Strasbourg, 09 2010, http://hal.inria.fr/inria-00534748/en/.

[17] B. PRADELLE, P. CLAUSS, V. LOECHNER. *Adaptive Runtime Selection of Parallel Schedules*, Université de Strasbourg, 09 2010, http://hal.inria.fr/inria-00534723/en/.

References in notes

[18] F. AGAKOV, E. BONILLA, J. CAVAZOS, B. FRANKE, G. FURSIN, M. F. P. O'BOYLE, J. THOMSON, M. TOUSSAINT, C. K. I. WILLIAMS. *Using Machine Learning to Focus Iterative Optimization*, in "CGO '06: Proceedings of the International Symposium on Code Generation and Optimization", Washington, DC, USA, IEEE Computer Society, 2006, p. 295–305, http://dx.doi.org/10.1109/CGO.2006.37.

[19] R. BACK. *On the Correctness of Refinement Steps in Program Development*, University of Helsinki, 1978.

[20] J.-P. BANÂTRE, D. L. MÉTAYER. *The* GAMMA *Model and its Discipline of Programming*, in "Science of Computer Programming", 1990, vol. 15, n^{o} 1, p. 55-79.

[21] B. BARRAS, S. BOUTIN, C. CORNES, J. COURANT, J.-C. FILLIATRE, E. GIMENEZ, H. HERBELIN, G. HUET, C. MUNOZ, C. MURTHY, C. PARENT, C. PAULIN-MOHRING, A. SAIBI, B. WERNER. *The Coq Proof Assistant Reference Manual : Version 6.1*, 1997.

[22] M. M. BASKARAN, U. BONDHUGULA, S. KRISHNAMOORTHY, J. RAMANUJAM, A. ROUNTEV, P. SADAYAPPAN. *A compiler framework for optimization of affine loop nests for GPGPUs*, in "ICS '08: Proceedings of the 22nd annual international conference on Supercomputing", New York, NY, USA, ACM, 2008, p. 225–234, http://doi.acm.org/10.1145/1375527.1375562.

[23] C. BASTOUL. *Code Generation in the Polyhedral Model Is Easier Than You Think*, in "PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques", Juan-les-Pins, France, 2004, p. 7–16, http://hal.ccsd.cnrs.fr/ccsd-00017260.

[24] Y. BERTOT, B. GRÉGOIRE, X. LEROY. *A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis*, in "TYPES 2004", 2004, p. 66-81.

[25] U. BONDHUGULA, A. HARTONO, J. RAMANUJAM, P. SADAYAPPAN. *A practical automatic polyhedral parallelizer and locality optimizer*, in "PLDI '08", ACM, 2008, p. 101–113, pluto-compiler.sourceforge.net, http://doi.acm.org/10.1145/1375581.1375595.

[26] L. BOUGÉ, Y. L. GUYADEC, G. UTARD, B. VIROT. *A Proof System for a Simple Data-Parallel Programming Language*, in "IFIP WG 10.3, Applications in Parallel and Distributed Computing", Caracas (Venezuela), North-Holland, April 1994.

[27] M. BRIDGES, N. VACHHARAJANI, Y. ZHANG, T. JABLIN, D. I. AUGUST. *Revisiting the Sequential Programming Model for Multi-Core*, in "MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture", Washington, DC, USA, IEEE Computer Society, 2007, p. 69–84, http://dx.doi.org/10.1109/MICRO.2007.35.

[28] M. BURTSCHER, I. GANUSOV, S. J. JACKSON, J. KE, P. RATANAWORABHAN, N. B. SAM. *The VPC Trace-Compression Algorithms*, in "IEEE Trans. Comput.", 2005, vol. 54, n^{o} 11, p. 1329–1344.

[29] D. CACHERA, D. PICHARDIE. *Embedding of Systems of Affine Recurrence Equations in Coq*, in "Proc. of 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)", Lecture Notes in Computer Science, Springer-Verlag, 2003, n^{o} 2758, p. 155–170.

[30] K. CHANDY, J. MISRA. *Parallel Program Design: A Foundation*, Addison Wesley, 1988.

[31] M. CINTRA, D. R. LLANOS. *Design Space Exploration of a Software Speculative Parallelization Scheme*, in "IEEE Trans. Parallel Distrib. Syst.", 2005, vol. 16, n^{o} 6, p. 562–576, http://dx.doi.org/10.1109/tpds.2005.69.

[32] P. FEAUTRIER. *Dataflow analysis of scalar and array references*, in "International Journal of Parallel Programming", 1991, vol. 20, n^{o} 1, p. 23–53.

[33] P. FEAUTRIER. *Some efficient solutions to the affine scheduling problem, Part 1 : one dimensional time*, in "International Journal of Parallel Programming", 1992, vol. 21, n^{o} 5, p. 313–348.

[34] P. FEAUTRIER. *Some efficient solutions to the affine scheduling problem, Part 2 : multidimensional time*, in "International Journal of Parallel Programming", 1992, vol. 21, n^{o} 6.

[35] P. FEAUTRIER. *Automatic Parallelization in the Polytope Model*, in "The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications", Springer-Verlag, 1996, p. 79–103.

[36] X. FENG, Z. SHAO, Y. DONG, Y. GUO. *Certifying low-level programs with hardware interrupts and preemptive threads*, in "SIGPLAN Not.", 2008, vol. 43, n^{o} 6, p. 170–182, http://dx.doi.org/10.1145/1379022.1375603.

[37] C. FLANAGAN, S. N. FREUND, J. YI. *Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs*, in "PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation", New York, NY, USA, ACM, 2008, p. 293–303, http://dx.doi.org/10.1145/1375581.1375618.

[38] P. GERNER. *La sémantique des directives au compilateur : application au parallélisme de données*, Université Louis Pasteur, 2002.

[39] P. GERNER, É. VIOLARD. *A Theoretical Framework of Data Parallelism and Its Operational Semantics*, in "Euro-Par 2000", LNCS, Springer, 2001, vol. 1900, p. 668–677.

[40] E. P. GRIBOMONT. *Stepwise refinement and concurrency: the finite-state case*, in "Sci. Comput. Program.", 1990, vol. 14, n^o 2-3, p. 185–228, http://dx.doi.org/10.1016/0167-6423(90)90020-E.

[41] M. HALL, D. PADUA, K. PINGALI. *Compiler research: the next 50 years*, in "Commun. ACM", 2009, vol. 52, n^o 2, p. 60–67, http://doi.acm.org/10.1145/1461928.1461946.

[42] A. HOBOR, A. W. APPEL, F. Z. NARDELLI. *Oracle Semantics for Concurrent Separation Logic*, in "ESOP", 2008, p. 353-367.

[43] M. KULKARNI, K. PINGALI, B. WALTER, G. RAMANARAYANAN, K. BALA, L. P. CHEW. *Optimistic parallelism requires abstractions*, in "PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation", New York, NY, USA, ACM, 2007, p. 211–222, http://doi.acm.org/10.1145/1250734.1250759.

[44] J. LARUS, C. KOZYRAKIS. *Transactional memory*, in "Commun. ACM", 2008, vol. 51, n^o 7, p. 80–88.

[45] E. A. LEE. *The Problem with Threads*, in "Computer", 2006, vol. 39, n^o 5, p. 33–42, http://dx.doi.org/10.1109/MC.2006.180.

[46] C. LENGAUER. *Loop Parallelization in the Polytope Model*, in "Parallel Processing Letters", 1994, vol. 4, n^o 3.

[47] X. LEROY. *Formal verification of a realistic compiler*, in "Communications of the ACM", July 2009, To appear.

[48] X. LEROY. *The Compcert verified compiler, software and commented proof*, January 2010, http://compcert.inria.fr.

[49] S.-W. LIAO, A. DIWAN, R. P. BOSCH, A. GHULOUM, M. S. LAM. *SUIF Explorer: an interactive and interprocedural parallelizer*, in "PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming", New York, NY, USA, ACM, 1999, p. 37–48, http://doi.acm.org/10.1145/301104.301108.

[50] V. LOECHNER, B. MEISTER, P. CLAUSS. *Data Sequence Locality: a Generalization of Temporal Locality*, in "Euro-Par 2001", Manchester, UK, Springer, 2001.

[51] V. LOECHNER, C. MONGENET. *Communication Optimization for Affine Recurrence Equations using Broadcast and Locality*, in "International Journal of Parallel Programming", 2000, vol. 28, n^o 1.

[52] C.-K. LUK, R. COHN, R. MUTH, H. PATIL, A. KLAUSER, G. LOWNEY, S. WALLACE, V. J. REDDI, K. HAZELWOOD. *Pin: building customized program analysis tools with dynamic instrumentation*, in "PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation", New York, NY, USA, ACM, 2005, p. 190–200, http://doi.acm.org/10.1145/1065010.1065034.

[53] K. F. MOORE, D. GROSSMAN. *High-level small-step operational semantics for transactions*, in "POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages", New York, NY, USA, ACM, 2008, p. 51–62, http://dx.doi.org/10.1145/1328438.1328448.

[54] T. MOSELEY, D. A. CONNORS, D. GRUNWALD, R. PERI. *Identifying potential parallelism via loop-centric profiling*, in "CF '07: Proceedings of the 4th international conference on Computing frontiers", ACM, 2007, p. 143–152.

[55] G. C. NECULA. *Translation validation for an optimizing compiler*, in "SIGPLAN Not.", 2000, vol. 35, n^o 5, p. 83–94, http://doi.acm.org/10.1145/358438.349314.

[56] A. PNUELI, O. SHTRICHMAN, M. SIEGEL. *The Code Validation Tool (CVT) - Automatic verification of code generated from synchronous languages*, in "Software Tools for Technology Transfer", 1998, vol. 2.

[57] L.-N. POUCHET, C. BASTOUL, A. COHEN, J. CAVAZOS. *Iterative optimization in the polyhedral model: part II, multidimensional time*, in "PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation", New York, NY, USA, ACM, 2008, p. 90–100, http://doi.acm.org/10.1145/1375581.1375594.

[58] L.-N. POUCHET, C. BASTOUL, A. COHEN, N. VASILACHE. *Iterative Optimization in the Polyhedral Model: part I, One-Dimensional Time*, in "CGO '07: Proceedings of the International Symposium on Code Generation and Optimization", Washington, DC, USA, IEEE Computer Society, 2007, p. 144–156, http://dx.doi.org/10.1109/CGO.2007.21.

[59] G. D. PRICE, J. GIACOMONI, M. VACHHARAJANI. *Visualizing potential parallelism in sequential programs*, in "PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques", New York, NY, USA, ACM, 2008, p. 82–90, http://doi.acm.org/10.1145/1454115.1454129.

[60] E. RAMAN, N. VACHHARAJANI, R. RANGAN, D. I. AUGUST. *Spice: speculative parallel iteration chunk execution*, in "CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization", New York, NY, USA, ACM, 2008, p. 175–184, http://doi.acm.org/10.1145/1356058.1356082.

[61] L. RAUCHWERGER, D. PADUA. *The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization*, in "IEEE Trans. Parallel Distrib. Syst.", 1999, vol. 10, n^o 2, p. 160–180, http://dx.doi.org/10.1109/71.752782.

[62] R. SEGHIR. *Méthodes de dénombrement de points entiers de polyèdres et applications à l'optimisation de programmes*, Université de Strasbourg, December 2006.

[63] T. SHERWOOD, E. PERELMAN, G. HAMERLY, B. CALDER. *Automatically characterizing large scale program behavior*, in "ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems", New York, NY, USA, ACM, 2002, p. 45–57, http://doi.acm.org/10.1145/605397.605403.

[64] J. SMITH, R. NAIR. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[65] C. TIAN, M. FENG, V. NAGARAJAN, R. GUPTA. *Copy or Discard Execution Model For Speculative Parallelization On Multicores*, in "IEEE/ACM 41st International Symposium on Microarchitecture, MICRO 41", November 2008, p. 330-341.

[66] J.-B. TRISTAN, X. LEROY. *Formal verification of translation validators: a case study on instruction scheduling optimizations*, in "SIGPLAN Not.", 2008, vol. 43, n^{o} 1, p. 17–27, http://dx.doi.org/10.1145/1328897.1328444.

[67] J.-B. TRISTAN, X. LEROY. *Verified Validation of Lazy Code Motion*, in "Programming Language Design and Implementation 2009", ACM Press, 2009, To appear.

[68] É. VIOLARD, S. GENAUD, G.-R. PERRIN. *Refinement of Data Parallel Programs in PEI*, in "Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi", London, UK, UK, Chapman & Hall, Ltd., 1997, p. 107–131.

[69] A. WELC, S. JAGANNATHAN, A. HOSKING. *Safe futures for Java*, in "OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications", New York, NY, USA, ACM, 2005, p. 439–453, http://doi.acm.org/10.1145/1094811.1094845.

[70] Q. WU, O. MENCER. *Evaluating Sampling Based Hotspot Detection*, in "International Conference on Architecture of Computing Systems, ARCS", March 2009.

[71] B. XIN, W. N. SUMNER, X. ZHANG. *Efficient program execution indexing*, in "PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation", New York, NY, USA, ACM, 2008, p. 238–248, http://doi.acm.org/10.1145/1375581.1375611.