Contributions to the Design of Reliable and Programmable High-Performance Systems: Principles, Interfaces, Algorithms and Tools

Albert Cohen

THÈSE d'HABILITATION à DIRIGER des RECHERCHES

Spécialité : Informatique

présentée par

Albert COHEN

pour obtenir l'HABILITATION à DIRIGER des RECHERCHES de l'UNIVERSITÉ PARIS-SUD 11

Sujet :

Contributions à la conception de systèmes à hautes performances, programmables et sûrs : principes, interfaces, algorithmes et outils

Contributions to the Design of Reliable and Programmable High-Performance Systems: Principles, Interfaces, Algorithms and Tools

Soutenue le 23 mars 2007 devant le jury composé de :

Nicolas	HALBWACHS	Rapporteur
François	IRIGOIN	Rapporteur
Lawrence	RAUCHWERGER	Rapporteur
Alain	DARTE	Examinateur
Marc	DURANTON	Examinateur
Olivier	TEMAM	Examinateur
Paul	FEAUTRIER	Membre Invité

Thèse d'Habilitation préparée au sein de l'équipe ALCHEMY
INRIA Futurs et LRI, UMR 8623 CNRS et Université Paris-Sud 11

Remerciements

À Isabelle, Mathilde et Loïc,
si proches et si souvent inaccessibles.

Je remercie les membres de mon jury pour avoir accepté de rapporter sur mon travail. Leur apport critique est précieux : il répond à une rare occasion de rassembler un ensemble significatif de résultats, et de tracer ainsi des pistes d'approfondissement ou d'exploration prioritaires. Je leur en suis d'autant plus reconnaissant, que leur rôle n'est pas plus confortable que celui du candidat dans l'exercice mandarinal de l'habilitation à diriger des recherches. Loin de moi l'idée de contester le principe originel de l'exercice, étape importante incitant tout chercheur à réaliser et publier une synthèse de ses travaux. Mais dans sa forme hexagonale contemporaine, l'exercice est dénaturé au point de ne refléter que des motivations mandarinales et jacobines beaucoup moins louables. Je m'efforcerai donc un jour de consacrer les mois nécessaires à l'écriture d'un livre de référence. Pas aujourd'hui, c'est bien trop tôt.

Je salue chaleureusement tous mes camarades de remue-méninges et de labeur développatoire ou TEXnique, d'hier et d'aujourd'hui, de toujours et d'un jour... sans qui peu de choses auraient été possibles. Ils se reconnaîtront, qu'ils fussent étudiants éclairés ou déprimés, ingénieurs virtuoses ou fatigués, chercheurs en herbe ou académiciens : leur travail et leur stimulation constantes sont le moteur essentiel de toute mon activité passée et future.

Je saisis l'opportunité de saluer l'un de ces premiers camarades, Jean-François Collard, moteur de la compilation par instances, mentor aux préceptes éclairés, héritier des grands sages fondateurs (et néanmoins collègues) Paul Feautrier et Luc Bougé que je salue également. Je remercie également Christine Eisenbeis pour m'avoir ouvert les portes de l'INRIA, et surtout, pour m'avoir constamment encouragé à dépasser les frontières du cloisonnement de la connaissance académique. Enfin, je ne trouve pas les mots pour saluer l'influence déterminante d'Olivier Temam, dont la vision, l'enthousiasme, le recul et la générosité m'ont accompagné dans la construction d'une stratégie de recherche que je crois cohérente et pertinente.

Je tiens également à tirer mon chapeau aux instituts de recherche et d'enseignement supérieur français et étrangers qui ont soutenu ce travail, et à travers eux, à tous mes collègues “accompagnateurs et accompagnatrices de la recherche”. Je suis particulièrement débiteur de l'INRIA, établissement permettant l'expression de projets scientifiques et technologiques ambitieux dans un environnement de liberté, de souplesse et de confort qu'il se doit de protéger, alors que de nombreux instruments de recherche et d'enseignement supérieur, en France et en Europe, sombrent dans l'incapacité, l'incompréhension et l'abandon.

Dedicated to a Brave GNU World

`http://www.gnu.org`

`Albert.Cohen@inria.fr`

Contents

List of Figures

Chapter 1

Introduction

It is an exciting time for high-performance and embedded computing research. The exponential growth of computing resources enters dangerous waters: the physics of silicon-based semiconductors is progressively putting Moore's law to an end. The close demise of this empirical law threatens the domination of 40-years old incremental research in the compilation of imperative languages and (super-)scalar von Neumann architectures. This represents an unprecedented challenge for the whole computer research and industry. As a positive side effect, the spectrum and depth of applied research broadens to new horizons, allowing to revisit scientific and technical areas once doomed too disruptive. This is a great opportunity for conducting fundamental research while maximizing the potential impact on actual computing systems.

In more immediate terms, these challenges are associated with the crackling of the von Neumann computing paradigm. The all-time dominant trend has been to push scalar architectures towards higher operating frequencies, showing secondary interest for power consuption, and hiding spatial concerns behind ever-increasing complexity. Ten years algo, this trend started to crackle in power- and area-efficient embedded systems; its collapse is complete with the now ubiquitous on-chip multi-processor architectures. Nevertheless, despite decades of academic and commercial attention, parallel computing is nowhere close to the maturity and accessibility of single-threaded programming and software engineering practices.

- The design of concurrent hardware (on-chip or multi-chip) did make tremendous progress: only the emergence of post-semiconductor technologies will, in a forseable future, disrupt the state of the art.
- Operating systems for shared and distributed memory architectures also made a giant leap (both the kernel and application-support libraries): just consider the scalability and portability of GNU/Linux, from 1024 Itanium 2 NUMA (SGI Altix) to heterogeneous system-on-chip architectures based on a variety of ISAs (ARM 7–11, SH 4, ST 231, etc.) and interconnection networks; yet the situation seems far from ideal in terms of efficiency (especially, dealing with fine grain, tightly coupled threads) and reliability (although the kernel may be closely looked upon by concurrency experts).
- The picture is much bleaker for the programming languages and compilers. Just ask yourself the question: what parallel programming language/model would you recommend and teach to mainstream application developpers? It is vain to answer there is no such thing as a universal parallel computing theory. Even restricting to a specific domain, and even recognizing the need for programmers to adopt a different state of mind when designing parallel software, it is hard to pick-up a satisfactory answer from the state-of-the-art.

Position of our contributions. Unlike most researchers in applied high-performance computing, we do not believe the main problem comes from concurrency itself. Despite the lack of a unifying model, parallel computing does not lack well understood semantics, syntaxes and and concurrency-aware compilation schemes. Emerging from the data-flow computing model [Kah74], from the reactive control theory [Cas01] and from synchronous programming languages [BCE$^+$03], we believe the *data-flow synchronous* model [CP96, CGHP04, CDE$^+$06] has the highest potential for high-performance, general-purpose and embedded computing:

- it expresses regular and irregular concurrency in a *compositional* (some say modular) and *deterministic* fashion;
- it may serve as an *intermediate language* for computation, control and data-centric parallel computing, synthesis, to generate multi and single-threaded scalar code as well as synchronous and asynchronous circuits;

- on demand, it may enforce *non-functional properties* (liveness, boundedness of memory, real-time) of the generated software/hardware *by construction*;
- our ongoing research encourages us to believe it is *overhead-free*, meaning that a portable synchronous dataflow program can be transformed into a target-specific one expliciting all the spatial and dynamic aspects of the underlying hardware/software layers, not relying on any hidden runtime support of inspection.

If concurrency is not, per se, the main challenge, then why is parallel computing not mainstream? Our answer comes from the in-depth analysis of two apparently simpler problems:

- architecture-aware optimization of single-threaded imperative programs,
- and maximizing the compute-density of explicitely parallel streaming applications.

We addressed these two problems for 6 years, studying both principled and applied viewpoints, and following a theoretical thesis on automatic parallelization [Coh99] (focusing on the extraction of static forms of parallelism in both regular and irregular programs). One important lesson is that resource management is by far the most complex and combinatorial task for the compiler, and the most misunderstood, hard to generalize and low-productivity activity for the system designer.

This lesson strongly influenced our ongoing research and long-term strategy.

1. We acknowledge the spatial distribution of the hardware, the combined complexity of underlying hardware/software layers, and the dominance of the resource mapping problem in current and future high-performance systems. This leads to the design of mapping-aware concurrent intermediate representations — the long term goal beyond our proposed *n*-synchronous Kahn networks — and this also motivates our empirical, iterative and adaptive optimization work.
2. We ought to learn, understand, synthesize and teach the rationale behind relevant research in computer architecture, runtime/operating systems, backend and high-level compilation, software engineering, and programming languages.
3. We aim to contribute to the design of computing systems that are simultaneously **scalable**, **productive**, **efficient** and **reliable**; in the following, these four goals will be instanciated in language design, compiler algorithms and compiler internals.

We do not expect these four goals will be simultaneously satisfied anytime soon, even with a larger and more diverse community recently addressing them. Indeed, parallel computing pioneers were facing somewhat simpler problems (mostly because of the elitist environment of engineers interested in parallel computers), and although some of the most brilliant computer scientists and engineers contributed to this fertile research area, the current state of the art is quite disappointing.

Nobody will argue against the importance of the first two goals. Yet it is unfortunate that many researchers do not consider efficiency (in power and space) and reliability (either by construction or through tolerant detection and replay mechanisms) as critical elements of a computing system. We do, because embedded systems — high-performance ones in particular — will likely become the dominant drive for computing research and engineering in the future, and because it is dangerous to ignore that complex concurrent systems are plagued with the nastiest bugs.

When studying scalability and efficiency, we consider the specific angle of architecture-aware code generation and optimization, targetting language designs that are both portable (the productivity goal) and overhead-free. We do not ignore the importance of higher-level programming models and the associated challenges with abstraction-penalty removal, but do not specifically address them.

When studying productivity, we only deal with the primitive constructs of intermediate languages and compiler internals, although we do check that all options are left open for more abstract language designs and software engineering to take advantage of them.

When studying relyability, we focus on the design and on the concurrency-induced problems. We also recognize the importance of fault detection and tolerance studies (putting speculative and transactionnal approaches into this body of work), but abstract them away, assuming the relevant mechanisms are present when needed.

Let us synthesize our interests and dedication. We work on the design and compilation of **intermediate language** layers, aiming for the satisfaction of the four abovementioned goals **by construction**,[1], operating at the finest possible level of the program semantics, the so-called **instancewise** level. Needless to tell this strategy may not bring the fastests results; we hope, however, it may contribute some of the most impactful in the long term.

[1] Rather than “passing the hot potato” to the next layer, which leads to diminishing returns.

Structure of this manuscript. This introductory chapter summarizes the state of the problem and the state of the art. It also surveys our research approach to *scalbility*, *productivity*, *efficiency* and *reliability* issues in programming and compiling for high-performance systems.

The three following chapters match the three technical areas where our contributions are well identified, and develop introductory and foundational material.

The last chapter discusses research perspectives that arise naturally from recent research partially covered in this manuscript.

The spinal column of this work is called *instancewise compilation* and will be described momentarily. It drives our contributions into four complementary aspects of the design of *reliable and programmable, high-performance systems*: the *principles*, the *algorithms*, the *interfaces* and the *tools*. We try to view these four aspects as equally important, justifying our empirical work and infrastructure developments (mostly in GCC and in polyhedral compilation technology) through contributions to the understanding of the deeper scientific problems, computing principles and algorithms.

1.1 Optimization Problems for Real Processor Architectures

Because processor architectures are increasingly complex, it has become practically impossible to embed accurate machine models within compilers. As a result, compiler efficiency tends to decrease with every improvement of processor *sustained* performance. To address this challenge, several research work on iterative, feedback-directed optimization [OKF00, FOK02, CST02] have proven the potential of iterative optimization. The goal of our research (and the resulting optimization process) is to address some of the *practical* issues that hinder the effective application of iterative optimization. Feedback-directed techniques [KKOW00, FOK02, OKF00, CST02] are currently limited to finding appropriate program transformation parameters, such as tile size, unroll factor, padding size, rather than the program transformation themselves, let alone compositions of program transformations; however, several recent work have outlined that complex and variable compositions of program transformations can be necessary to reach high performance [YLR$^+$03, PTV02, PTCV04, CGP$^+$05], beyond the rigid sequence of program transformations embedded in static compilers. How can we find a proper composition of program transformations within such a huge search space? Currently, searching is restricted to a few optimizations, and even then, it usually requires several hundreds of runs using genetic algorithms or other operations research heuristics [KKOW00, CST02, SAMO03].

To better understand the potential and limitations of iterative optimization, Parello et al. adopted a bottom-up approach to the architecture complexity issue. We recognize this work as a milestone on the road towards a new generation of optimization methodologies and optimizing compilers compatible with the complexity and variability of the hardware. Therefore, although we played a late and minor role in this work [PTCV04], it consitutes an important baseline for our research. Parello's bottom-up approach is the following: assuming we know everything about the behavior of the program on the target processor (extensive dynamic analysis), what can we do to improve its performance? An extensive analysis of programs behaviors on a complex processor architecture led to the design of a systematic and iterative optimization methodology.

The approach takes the form of a decision tree which guides the optimization process. Each branch of the tree is a sequence of analysis/decision steps based on run-time metrics (dynamic analysis), called *performance anomalies*, and a branch leaf is one or a few localized program transformation suggestions. An iteration of the optimization process is equivalent to walking down one branch. After the corresponding optimization has been applied, the program is run again, new statistics are gathered, the process starts again at the tree top and a new branch is followed. Progressively, the process builds a sequence (composition) of program transformations. The process repeats until further transformations do not bring any significant additional improvement. Of course, the process is just one of the many possible "walks" within a huge search space, but this walk is systematic; to a limited extent it provides an approach for whole-program optimization and it has been experimentally proved to yield significant performance improvement on SPEC benchmarks for the Alpha 21264 processor, beyond peak SPEC (best optimization flags for each benchmark [Spe], using HP's latest compiler), see Figure 1.1.

The process relies on the observation of tenths of different performance *anomalies*; some of these anomalies correspond to traditional statistics, e.g., data TLB misses, available from program counters, and others are slightly more elaborate performance indicators. They aim at enumerating and separating the different possible causes of performance loss. Why do we need such "performance anomalies" and what are they exactly?

The initial motivation was to *find the exact cause of any performance loss* during a program execution, in order to apply the appropriate program optimization. In an out-of-order superscalar processor like the Alpha 21264, a

	Peak SPEC	Methodology		Peak SPEC	Methodology
swim	1.00	1.61	galgel	1.04	1.39
wupwise	1.20	2.90	applu	1.47	2.18
apsi	1.07	1.23	mesa	1.04	1.42
ammp	1.18	1.40	equake	2.65	3.22
mesa	1.12	1.17	mgrid	1.59	1.45
fma3d	1.32	1.09	art	1.22	1.07

Figure 1.1: Speedup for 12 SPEC CPU2000 fp benchmarks

performance loss occurs when, at a given cycle, the maximum number of instructions cannot be committed (11 in this case). Determining the cause of the performance loss means understanding why a given (or several) instruction(s) could not be committed. Determining the "real" cause for an instruction stall can be a very difficult task in such a processor because performance effects can propagate over a large number of cycles [DHW+97]: a data cache miss can slow down an arithmetic operation, which in turn has a resource conflict with another arithmetic operation, which in turn delays an address computation…so that the instruction at the source of the performance loss may have left the pipeline many cycles before, and there are often multiple intertwined causes.

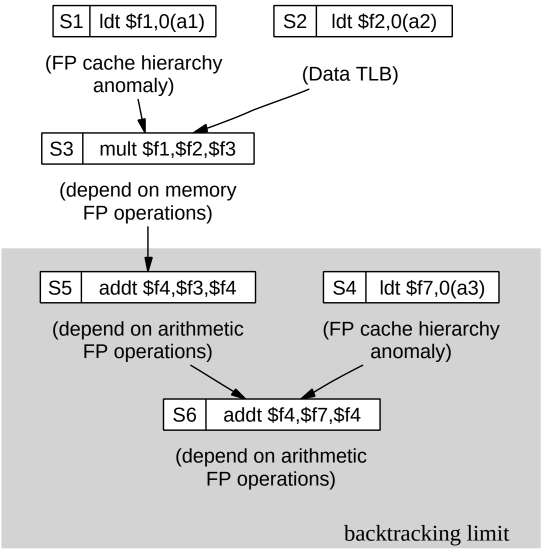

S1 ldt $f1,0(a1)
S2 ldt $f2,0(a2)
(FP cache hierarchy anomaly)
(Data TLB)
S3 mult $f1,$f2,$f3
(depend on memory FP operations)
S5 addt $f4,$f3,$f4
S4 ldt $f7,0(a3)
(depend on arithmetic FP operations)
(FP cache hierarchy anomaly)
S6 addt $f4,$f7,$f4
(depend on arithmetic FP operations)
backtracking limit

Figure 1.2: Bounded backward slice

A straightforward, local solution consists in monitoring key hardware components, considering a performance loss *may* occur as soon as a hardware component is not performing *at full capacity*. More precisely, to characterize the performance loss induced by a given instruction, one may restrict backtracking to the parent instructions only in the data-flow graph. And the selected program transformations will target that particular performance loss whether it is only a symptom or a cause. For instance in Figure 1.2: the analysis would naturally start with instruction *S*6 at the bottom of the data-flow tree, and it would be limited to its parents *S*4 and *S*5 in the grey areas.

This methodology, captured in a decision tree, iterates *dynamic analysis* phases of the program behavior — using a cycle-accurate simulator or hardware counters — *decision* phases to choose the next analysis or transformation to perform, and program transformation phases to address a given performance issue. After each transformation, the performance is measured on the real machine to evaluate the actual benefits/losses, then a new analysis phase is needed to decide whether it is worth iterating the process and applying a new transformation. Though this optimization process is manual, it is also *systematic* and iterative, the path through the decision tree being guided by increasingly detailed performance metrics. Except for precisely locating target code sections and checking the legality of program transformations, it could almost perform automatically.

From a program transformation point of view, our methodology results in a structured sequence of transformations applied to various code sections. The first result is that such transformation sequences are out of reach of current compiler technology [GVB+06]. Even worse, although particularly complex already, these sequences are only the premises of the real optimizations that will be needed on future architectures with multiple levels of parallelism, heterogeneous computing resources, explicit management of the communication network topology, and non-deterministic run-time adaptation systems. Beyond complexity and unpredictability, this example also shows how important are extensibility (provisions for implementing new transformations) and debugging support (static and/or dynamic).

Essentially, the research results surveyed in the following sections are as many coordinated attempts to avoid the diminishing returns associated with incremental improvements to current compilation and programming approaches.

1.2 Going Instancewise

Most programming languages closely reflect the inductive nature of the main Church-equivalent computing models and bring *abstraction as a feature for programmer's comfort*. We are interested in compilers that operate on semantically richer program abstractions, with statically tractable algebraic properties (i.e., closed mathematical forms, representations in Presburger arithmetic or decidable automata-theoretic classes).

Whether denotational, operational or axiomatic, program semantics assigns "meaning" to a *finite set of syntactic elements* — statements or variables — using inductive definitions. When designing a static analysis or transformation framework to reason about programs, it is very natural to attach static properties to this finite set of syntactic elements. Indeed, in many situations, semantical program manipulations operate locally on the inductive definitions associated with each program expression.

For the more sophisticated analyses and transformations, being so closely related to the inductive definitions of the semantics may not be practical. Another approach, sometimes referred to as *constraint-based* in the static analysis context [NNH99], consists in operating on an abstraction more or less decoupled from the natural inductive semantics; typically, a system of *constraints* that characterize the static property or the program behavior of interest.

For example, constant propagation [ASU86] amounts to computing a property of a variable v at a statement s, asking whether v has some value v before s executes. It is quite natural to formalize constant propagation as a type system, in a data-flow setting or in abstract interpretation. But let us now consider another static analysis problem that may be seen as an extension of constant propagation: *induction variable recognition* [GSW95] captures the value of some variable v at a statement s as a function f_{v} of the number of times s has been executed. In other words, it captures v as a *function of the execution path* itself. Of course, the value of a variable at any stage of the execution is a function of the initial contents of memory and of the execution path leading to this stage. For complexity reasons, the execution path may not be recoverable from memory. In the case of induction variables, we may assume the number of executions of s is recorded as a genuine loop counter. From such a function f_{v} for s, we can discover the other induction variables using analyses of linear constraints [CH78], but such syntactically bound approaches will not easily cope with the calculation of function f_{v} itself.

In the following, we will qualify as ***instancewise*** *any compilation method operating on finitely presented functions of the infinite set of runtime control points.*

Historically, the instancewise approach derived from loop-restructuring compiler frameworks [Wol96] aiming at a large spectrum of optimizations: vectorization, instruction-level, thread-level or data parallelism, scheduling and mapping for automatic parallelization, locality optimization, and many others [PD96, AK02]. The associated loop-nest analyses and representations share a common principle: they characterize static properties as *functions of run-time control points (infinite or unbounded)* and *not* as *functions of syntactic program elements (finite)*. Loop-restructuring compilers effectively operate at a higher level of understanding of the program behavior, decoupling program reasoning from the natural inductive semantics of most programming languages.

At this point, we are forced to reexamine the principles of traditional compilation methods in a wider world of abstract domains without fix-point calculations where loops and recursive functions may often be manipulated with maximal accuracy. For example, abstract interpretation [CC77] is certainly not the only way to use formal abstraction and concretization principles (as Galois connections or insertions) in compilation: the need to effectively resort to a form of static *interpretation* is only the translation of the inductive, fix-point based program semantics. When operating in a constraint-based representation of the program and its properties, this interpretation may be iteration-less [Muc97, WCPH01, PCS05] (no need to compute a fix point, as in many SSA-based analyses) or even resort to operation research algorithms thoroughly alien to program interpretation, including linear programming, constraint solving, and all sorts of empirical methods [Bar98, Fea92]. The work of Creusillet [Cre96] is one of the rare instancewise analyses to resort to abstract interpretation, the reason lying in its interprocedural nature.

Back to instancewise compilation, Figure 1.3 shows a synthetic example where an array A is initialized in a loop nest and read in a recursive procedure. The footprint of reads to A in procedure `line` is a “chess-board”: only elements `A[i][j]` such that $i+j$ is an even number are read in the procedure. This observation leads to a simple optimization: half of the dynamic assignments to A in the loop nest are useless, they can be avoided through a simple transformation of the bounds and strides. We call this optimization *instancewise dead-code elimination*.

A typical static analysis technique to solve this kind of problems is called *array-region analysis* [CH78, Cre96]. However, because the “chess-board” footprint is not a convex polyhedron, all the array-region analyses we are aware of will fail on this example. In theory, recovering such precise information seems possible by abstract interpretation, provided the widening operator for $\mathbb{Z}$-polyhedra (also called lattice polyhedra) can handle some level of non-convexity [Sch86, LW97], which is not the case in the current state of the art [BRZ03]. In addition, such precision may only be achieved by a *context-sensitive* analysis.

```
int A[10][10];

void line (int i, int j) {
  ... = A[i][j];
  if (j<10) line (i, j+2)
}

int main () {
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
      A[i][j] = ...;
  for (i=0; i<10; i+=2) {
    line(i, j);
    line(i+1, j+1);
}
```

Figure 1.3: Simple instancewise dead-code elimination

```
int A[10][10];

void line (int i, int j, int k, int l) {
  ... = A[i][j];
  if (j<10) line (k, l, i, j+2)
}

int main () {
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
      A[i][j] = ...;
  for (i=0; i<10; i+=2) {
    line(i, j, i+1, j+1);
}
```

Figure 1.4: More complex example

Figure 1.4 shows a slightly obfuscated version of the previous code: the procedure recursively swaps its arguments and only one the two initial calls remains in the loop. Although it may not seem obvious, this code has the same “chess-board” footprint as the previous one for reads to A, and the useless array assignments can still be removed. In this new form, it is of course much harder to imagine a precise enough widening operator. Recent techniques based on model-checking of push-down systems [EK99, EP00] would suffer from a similar limitation: although such techniques provide virtually unlimited context-sensitivity, operations on the abstract domain will incur necessary approximations and lead to a convex region instead of a “chess-board”.

In Chapter 2, we will describe a static analysis framework well fitted for this kind of problems. Instead of searching for more precision in the lattice, *it provides unlimited precision and context-sensitivity in the control domain, computing static properties as functions of an infinite set of run-time program points.* We apply this concept to the characterization of induction variables in recursive programs and to (elementwise, a.k.a. non-uniform) dependence analysis for trees and arrays in such programs. We prove that exact polyhedral dependence tests (e.g., the Omega [Pug91a] or PIP tests [Fea88b]) are special cases of a more general exact test for recursive programs with static restrictions on the guards of conditionals and loop bounds, and with a monoid abstraction of the structure of heap-allocated data. We provide polynomial or algorithms to compute these properties as rational transducers (two-tape automata), and practical algorithms to solve the dependence test itself (with exact and approximate versions, the problem being proven NP-complete). These results can be found in [Coh99, Ami04].

1.3 Navigating the Optimization Oceans

Recently, iterative optimization has become an increasingly popular approach for tackling the growing complexity of processor architectures. Bodin et al. [BKK+98], Barreteau et al.[2] [BBC+99] and Kisuki et al. [KKOW00] have initially demonstrated that exhaustively searching an optimization parameter space can bring performance improvements higher than the best existing static models, Cooper et al. [CST02] have provided additional evi-

[2] The title of the paragraph is an intended allusion to the OCEANS project, a milestone in this area.

dence for finding best sequences of various compiler transformations. Since then, recent studies [TVA05, FOK02, BFF05] demonstrate the potential of iterative optimization for a large range of optimization techniques.

Some studies show how iterative optimization can be used *in practice*, for instance, for tuning optimization parameters in libraries [WPD00, BACD97] or for building static models for compiler optimization parameters. Such models derive from the automatic discovery of the mapping function between key program characteristics and compiler optimization parameters; e.g., Stephenson et al. [SA05] successfully applied this approach to unrolling.

However, most other articles on iterative optimization take the same approach: several benchmarks are repeatedly executed with the same data set, a new optimization parameter (e.g., tile size, unrolling factor, inlining decision,...) being tested at each execution. So, while these studies demonstrate the *potential* for iterative optimization, few provide a *practical* approach for effectively applying iterative optimization. The issue at stake is: what do we need to do to make iterative optimization a reality? There are three main caveats to iterative optimization: quickly scanning a large search space, optimizing based on and across multiple data sets, and extending iterative optimization to complex composed optimizations beyond simple optimization parameter tuning.

We aim at the general goal of making iterative optimization a usable technique and especially focus on the first issue, i.e., how to speed up the scanning of a large optimization space. As iterative optimization moves beyond simple parameter tuning to composition of multiple transformations [FOK02, PTCV04, LF05, CGP$^+$05] (the third issue mentioned above), this search space can become potentially huge, calling for faster evaluation techniques. There are at least four possible ways to speeding up the search space scanning:

1. search more smartly by exploring points with the highest potential using genetic algorithms and machine learning techniques [CSS99, CST02, VAGL03, SAMO03, ACG$^+$04, MBQ02, HBKM03, SA05],

2. enhance the structure of the search space so that faster operation research algorithms with better understood mathematical properties can be applied [O'B98, CGP$^+$05],

3. or use the programmer's expertise to directly or indirectly drive the optimization heuristic [CHH$^+$93, LGP04, PTCV04, DBR$^+$05, CDG$^+$06],

4. evaluating multiple optimizations at runtime or reducing the duration of profile runs, effectively scanning more points within the same amount of time [FCOT05].

Speeding up the search has mostly focused on the first approach, while we have so far focused on the three other ones.

Enhancing the search-space structure. Optimizing compilers have traditionally been limited to systematic and tedious tasks that are either not accessible to the programmer (e.g., instruction selection, register allocation) or that the programmer in a high level language does not want to deal with (e.g., constant propagation, partial redundancy elimination, dead-code elimination, control-flow optimizations). Generating efficient code for deep parallelism and deep memory hierarchies with complex and dynamic hardware components is a completely different story: the compiler (and run-time system) now has to take the burden of much smarter tasks, that only expert programmers would be able to carry. In a sense, it is not clear that these new optimization and parallelization tasks should be called "compilation" anymore. Iterative optimization and machine learning compilation [KKOW00, CST02, LO04] are part of the answer to these challenges, building on artificial intelligence and operation research know-how to assist compiler heuristic. Iterative optimization generalizes profile-directed approach to integrate precise feedback from the runtime behavior of the program into optimization algorithms, while machine learning approaches provide an automated framework to build new optimizers from empirical optimization data. However, considering the ability to perform complex transformations, or complex sequences of transformations [PTV02, PTCV04], iterative optimization and machine learning compilation will fare no better than existing compilers on top of which they are currently implemented. In addition, any operation research algorithm will be highly sensitive to the structure of the search space it is traversing. E.g., genetic algorithms are known to cope with unstructured spaces but at a higher cost and lower scalability towards larger problems, as opposed to mathematical programming (e.g., semi-definite or linear programming) which benefit from strong regularity and algebraic properties of the optimization search space. Unfortunately, current compilers offer a very unstructured optimization search space. First of all, by imposing phase ordering constraints [Wol96], they lack the ability to perform long sequences of transformations. In addition, compilers embed a large collection of ad-hoc program transformations, but they are *syntactic* transformations, i.e., control structures are regenerated after each program transformation, sometimes making it harder to apply the next transformations, especially when the application of program transformations relies on pattern-matching techniques.

Clearly, there is a need for a compiler infrastructure that can apply complex and possibly long compositions of optimizing or parallelizing transformations, in a rich, structured search space.

We claim that existing compilers are ill-equipped to address these challenges, because of improper program representations and inappropriate conditioning of the search space structure.

In Chapter 3, we try to remedy to the lack of an algebraic structure in traditional loop-nest optimizers, as a small step towards bridging the gap between peak and sustained performance in future and emerging on-chip multiprocessors. Our framework facilitates the search for *compositions* of program transformations, relying on a unified representation of loops and statements [CGP$^+$05]. This framework improves on classical polyhedral representations [Fea92, Wol92, Kel96, LL97, AMP00, LLL01] to support a large array of useful and efficient program transformations (loop fusion, tiling, array forward substitution, statement reordering, software pipelining, array padding, etc.), as well as *compositions* (also in a mathematical sense) of these transformations. Compared to the attempts at expressing a large array of program transformations as matrix operations within the polyhedral model [Wol92, Pug91c, Kel96], the distinctive asset of our representation lies in the simplicity of the formalism to compose non-unimodular transformations across long, flexible sequences. Existing formalisms have been designed for black-box optimization [Fea92, LL97, AMP00], and applying a classical loop transformation within them — as proposed in [Wol92, Kel96, LO04] — requires a syntactic form of the program to anchor the transformation to existing statements. Up to now, the easy composition of transformations was restricted to unimodular transformations [Wol96], with some extensions to singular transformations [LP94].

The key to our approach is to clearly separate the four different types of actions performed by program transformations: modification of the iteration domain (loop bounds and strides), modification of the schedule of each individual statement, modification of the access functions (array subscripts), and modification of the data layout (array declarations). This separation makes it possible to provide a matrix representation for each kind of action, enabling the easy and independent composition of the different “actions” induced by program transformations, and as a result, enabling the composition of transformations themselves. Current representations of program transformations do not clearly separate these four types of actions; as a result, the implementation of certain compositions of program transformations can be complicated or even impossible. For instance, current implementations of loop fusion must include loop bounds and array subscript modifications even though they are only byproducts of a schedule-oriented program transformation; after applying loop fusion, target loops are often peeled, increasing code size and making further optimizations more complex. Within our representation, loop fusion is only expressed as a schedule transformation, and the modifications of the iteration domain and access functions are implicitly handled, so that the code complexity is exactly the same before and after fusion. Similarly, an iteration domain-oriented transformation like unrolling should have no impact on the schedule or data layout representations; or a data layout-oriented transformation like padding should have no impact on the schedule or iteration domain representations. Eventually, since all program transformations correspond to a set of matrix operations within our representation, searching for compositions of transformations is often (though not always) equivalent to testing different values of the matrices parameters, further facilitating the search for compositions. Besides, with this framework, it should also be possible to find and evaluate new sequences of transformations for which no static model has yet been developed (e.g., array forward substitution versus loop fusion as a temporal locality optimization).

Using the programmer’s expertise. Beyond a potential strategy for driving iterative optimization, the methodological work on bottom-up optimization outlined at the beginning of this section [PTCV04] has several immediate benefits. (1) It provides a manual optimization process that can be used by engineers; because this process is systematic, less expertise is required on the part of the engineer to optimize a program. (2) The decision tree formalizes the empirical expertise of engineers, and it is a way to pass this expertise, traditionally hard to teach, to new engineers or researchers. (3) Each branch actually defines a mapping between a given architecture performance issue and appropriate program transformations; this mapping is based on empirical expertise. (4) Beyond the optimization process, this empirical work also had the benefit of filtering which, among the many existing program transformations, bring the best benefits in practice.

Indeed, programmers of computationally intensive applications complain about the lack of efficiency of their machines — the ratio of sustained to peak performance — and the poor performance of optimizing compilers [WPD00]. Of course, they do not wait for research prototypes to become production-quality optimizers before attempting to improve the productivity of their manual, application-specific optimizations. In addition, no black-box compiler has ever compiled a matrix-matrix product code written in Fortran or C on a modern multi-core superscalar processor and reached performance levels close to hand-tuned mathematical libraries. There are fundamental reasons for such a disastrous situation:

- domain-specific knowledge unavailable to the compiler can be required to prove optimizations' legality or profitability [BGGT02, LBCO03];
- hard-to-drive transformations are not available in compilers, including transformations whose profitability is difficult to assess or whose risk of degrading performance is high, e.g., speculative optimizations [ACM+98, RP99];
- complex loop transformations do not compose well, due to syntactic constraints and code size increase [CGT04];
- some optimizations are in fact algorithm replacements, where the selection of the most appropriate code may depend on the architecture and input data [LGP04].

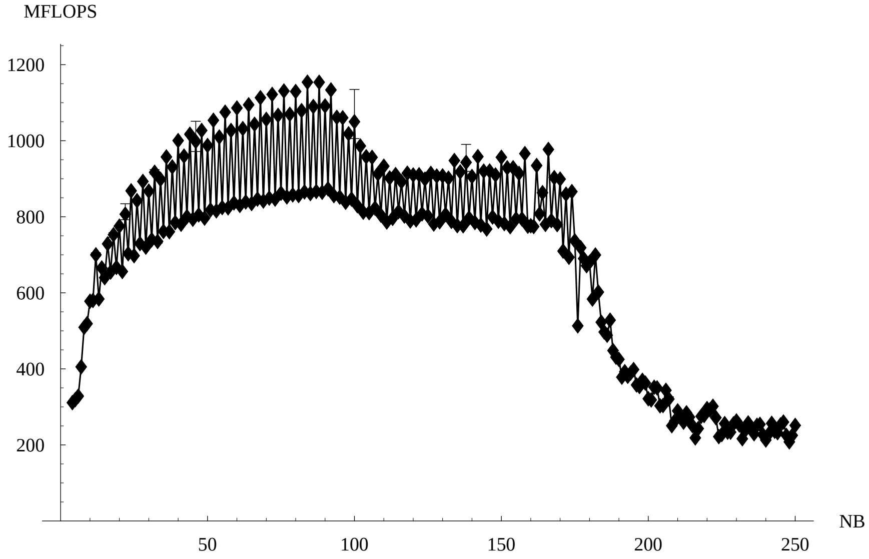


Figure 1.5: Influence of parameter selection

It is well known that manual optimizations degrade portability: the performance of a C or Fortran code on a given platform does not say much about its performance on different architectures. Several people have successfully addressed this issue, not by improving the compiler, but through the design of application-specific program generators, a.k.a. active libraries [VG98]. Such generators often rely on feedback-directed optimization to select the best generation strategy [Smi00], but not exclusively [YLR+03]. The most popular examples are ATLAS [WPD00] for dense matrix operations and FFTW [FJ98] for the fast Fourier transform. Such generators follow an iterative optimization scheme. In the case of ATLAS: an external control loop generates multiple versions of the optimized code, varying the optimization parameters such as the tile size of the blocked matrix product, loop unrolling factors, etc. These versions are benchmarked, and the empirical search engine selects the best parameters. Figure 1.5 shows the influence of the tile size of the blocked matrix product on performance on the AMD Athlon MP processor; as expected from the two-level cache hierarchy of the processor, three main intervals can be identified, corresponding to the temporal reuse of cached array elements; it is harder to predict and statically model the pseudo-periodic variations within each interval, due to alignment and associativity conflicts in caches [YLR+03].

Most of these generators use transformations previously proposed for traditional compilers, which fail to apply them for the aforementioned reasons. Conversely, optimizations often involve domain knowledge, from the specialization and interprocedural optimization of library functions [DP99, CK01] to application-specific optimizations such as algorithm selection [LGP04]. Recently, the SPIRAL project [PSX+04] investigated a domain-specific extension of such program generators, operating on a domain-specific language of digital signal processing formulas. This project is one step forward to bridge the gap between application-specific generators and generic compiler-based approaches, and to improve the portability of application performance.

We advocate for the use of generative programming languages and techniques, to support the design of such generic adaptive libraries by high-performance computing experts [CDG$^+$06]. We show that complex optimizations can be implemented in a type-safe, purely generative framework. We also show that peak performance is achievable through the careful combination of a high-level, multi-stage language — MetaOCaml [CTHL03] — with low-level code generation techniques.

We also show that combining generative techniques with *semantics-preserving transformations* is an even better solution, and may further improve the productivity of high-performance library developers [DBR$^+$05]. This approach can be opposed to introspective and reflective approaches in more expressive meta-programming frameworks: it provides the right abstractions and primitives for architecture-aware optimization while preserving the most important safety properties. We are planning further research in this area, in particular through an attempt to combine the rich algebraic structure of the polyhedral model with generative programming.

Scanning more points within the same amount of time. The principle of our approach is to improve the efficiency of iterative optimization by taking advantage of program *performance stability* at run-time. There is ample evidence that many programs exhibit phases [SPHC02, LSC05], i.e., program trace intervals of several millions instructions where performance is similar. What is the point of waiting for the end of the execution in order to evaluate an optimization decision (e.g., evaluating a tiling or unrolling factor, or a given composition of transformations) if the program performance is stable within phases or the whole execution? One could take advantage of phase intervals with the same performance to evaluate a different optimization option at each interval. As in standard iterative optimization, many options are evaluated, except that multiple options are evaluated within the same run.

Note that there are alternative ways to gather feedback data in a shorter time. Statistical simulation techniques [SPHC02] could potentially be adapted to compilation purposes, provided an optimization-agnostic checkpointing of a process state can be performed [GMCT03]. Alternatively, one may also use machine learning techniques to construct cost models automatically, and apply these cost models instead of performing a full profile run. These ideas are left for future work.

The main assets of our approach over previous techniques are simplicity and practicality. We show that a low-overhead performance stability/phase detection scheme is sufficient for optimization space pruning for loop-based floating point benchmarks. We also show that it is possible to search (even complex) optimizations at runtime without resorting to sophisticated dynamic optimization/recompilation frameworks. Beyond iterative optimization, our approach also enables one to quickly design self-tuned applications, significantly easier than manually tuned libraries.

Phase detection and optimization evaluation are respectively implemented using code instrumentation and versioning within the EKOPath compiler. Considering 5 self-tuned SPEC CPU2000 fp benchmarks, our space pruning approach speeds up iterative search by a factor of 32 to 962, with a 99.4% accurate phase prediction and a 2.6% performance overhead on average; we achieve speedups ranging from 1.10 to 1.72 [FCOT05].

1.4 Harnessing Massive On-Chip Parallelism

Future and emerging processor designs will integrate massive amounts of parallelism. This is a fact of the physics, essentially due to communication delays (currently, wire delays), and marginally (or temporarily) to power dissipation and architecture design issues.

With such architectures, performance scalability is the most immediate challenge. While these processors start invading all domains of computing, they will also put an end to the low expectations in terms of *efficiency*, *programmer productivity* and *reliability* — thinking of the functional correctness and fault tolerance — that are unfortunately common in parallel computing. Who wants to program cheap massively-parallel chips with methodologies and systems for worldwide grids? Beyond pure performance, it is well known that this evolution stresses productivity issues in the design of parallel systems, as emphasized by the DARPA HPCS program and the Fortress language initiative by Sun Microsystems [ACL$^+$06].

In addition, the rapid evolution of embedded system technology — favored by Moore's law and standards — is increasingly blurring the barriers between the design of safety-critical, real-time and high-performance systems. A good example is the domain of high-end video applications, where tera-operations per second (on pixel components) in hard real-time will soon be common in low-power devices. Parallel embedded computing make the parallel programming productivity issue even more challenging: no current framework is able to bring compositionality to explicit time and resource management while generating efficient parallel code from high-level distributed computing models.

Unfortunately, general-purpose architectures and compilers are not suitable for the design of real-time *and* high-performance (massively parallel) *and* low-power *and programmable* system-on-chip [CDC$^+$03]. Achieving a high compute density and still preserving programmability is a challenge for the choice of an appropriate architecture, programming language and compiler. Typically, thousands of operations per cycle must be sustained on chip, exploiting multiple levels of parallelism in the compute kernel, with tightly coupled operations, while enforcing strong real-time properties.

To address these challenges, we studied the synchronous model of computation [BCE$^+$03] which allows for the generation of custom, parallel hardware and software systems with *correct-by-construction structural properties*, including real-time and resource constraints. This model met industrial success for safety-critical, reactive systems, through languages like SIGNAL [BLJ91], LUSTRE (SCADE) [HCRP91] and ESTEREL [Ber00].

To enforce real-time and resources properties, synchronous languages assume a common clock for all registers, and an overall predictable execution layer where communications and computations can be proven to take less than a (physical or logical) clock cycle. Due to wire delays, a massively parallel system-on-chip has to be divided into multiple, asynchronous clock domains: the so called *Globally Asynchronous Locally Synchronous* (GALS) model [Cha84]. This has a strong impact on the formalization of synchronous execution itself and on the associated compilation strategies [LTL03].

Due to the complexity of high-performance applications and to the intrinsic combinatorics of synchronous execution, *multiple clock domains* have to be considered *at the application level as well* [CDE$^+$05]. This is the case for modular designs with separate compilation phases, and for a single system with multiple input/output associated with different real-time clocks (e.g., video streaming). It is thus necessary to compose independently scheduled processes. *Kahn Process Networks* (KPN) [Kah74] can accommodate for such a composition, compensating for the local asynchrony through unbounded blocking FIFO buffers. But allowing a global synchronous execution imposes additional constraints on the composition. We introduce the concept of *n-synchronous* clocks to formalize these concepts and constraints. This concept describes naturally the semantics of KPN with bounded, statically computable buffer sizes. This extension allows the modular composition of independently scheduled components with multiple periodic clocks satisfying a flow preservation equation, through the automatic inference of bounded delays and FIFO buffers. Our first results are detailed in Chapter 4.

Chapter 2

Instancewise Program Analysis

This chapter studies the extension of instancewise compilation techniques to imperative, first-order, well structured recursive programs. It focuses on static analysis and on the characterization of induction variables as closed form expressions over non-tail-recursive definitions.

Statementwise analysis. We use the term *statementwise* to refer to the classical type systems, data-flow analysis and abstract interpretation frameworks, that define and compute program properties at each program statement. A typical example is static analysis by abstract interpretation [CC77, Cou81, Cou96]: it relies on the *collecting semantics* to operate on a lattice of abstract properties. This restricts the attachment of properties to a *finite set of control points*. Little research addressed the attachment of static properties at a finer grain than syntactic program elements. Refinement of this coarse grain abstraction involves a previous *partitioning* [Cou81] of the *control points*: e.g., *polyvariant* analysis distinguishes the context of function calls, and *loop unfolding* virtually unrolls a loop several times. *Dynamic partitioning* [Bou92] integrates partitioning into the analysis itself. Control points can be extended with *call strings* (abstract call stacks) and *timestamps*, but ultimately rely on *k-limiting* [SP81, Har89] or *summarization* heuristics [RHS95] to achieve convergence. Although unbounded lattices have long been used to capture abstract properties [CH78, Deu94]), there was little interest in the computation of data-flow facts attached to an *unbounded set of control points*, following the seminal paper by Esparza and Knoop [EK99]. This approach is the closest to our work and a detailed comparison is provided in Section 2.7; it builds on model-checking of push-down systems to extend precision and context sensitivity, without sacrifying efficiency [EP00], but it ultimately results in the computation of data-flow properties attached to a *finite number of control points*.

Instancewise analysis. On the other hand, ad-hoc, constraint-based approaches to static analysis are able to compute program properties as *functions defined on an infinite (or unbounded) number of run-time control points*. The so-called *polytope model* encompasses most work on analysis and transformation of the (Turing-incomplete) class of *static-control programs* [Fea88a, PD96], roughly defined as nested loops with affine loop bounds and array accesses. An *iteration vector* abstracts the runtime control point corresponding to a given iteration of a statement. Program properties are expressed and computed for each vector of values of the surrounding loop counters. In general, the result of the analysis is a mapping from the infinite set of iteration vectors (the run-time control points) to an arbitrary (analysis-specific) vector space (e.g., dependence vector). Instead of iteratively merging data-flow properties, most analyses in the polytope model use algebraic solvers for the direct computation of symbolic relations: e.g., array dependence analysis uses integer linear programming [Fea88a]. Iteration vectors are quite different from time-stamps in control point partitioning techniques [Bou92]: they are multidimensional, lexicographically ordered, *unbounded*, and constrained by Presburger formula [Pug92].

First contribution. We introduce a general static analysis framework that uncompasses most ad-hoc formalisms for the fine grain analysis of loop nests and arrays in sequential procedural languages. Within this framework, one may *define, abstract and compute* program properties at an *infinite* number of *runtime control points*. Our framework is called *instancewise* and runtime points are further referenced as *instances*. We will formally define instances as *trace abstractions*, understood as iteration vectors extended to arbitrary recursive programs. The mathematical foundation for instancewise analysis is *formal language theory*: rational languages finitely represent infinite set of instances, and instancewise properties may be captured by rational relations [Ber79]. This paper goes far beyond our previous attempts to extend iteration vectors to recursive programs, for the analysis of arrays [CCG96, CC98, Coh99, Col02, ACF03] or recursive data structures [Fea98, Col02, Coh99].

Second contribution. Building on the instancewise framework, we extend the concept of *induction variables* to arbitrary recursive programs. This extension demonstrates the ability to characterize properties of programs as functions from an infinite set of run-time control points, beyond the restricted domain of Fortran loop nests. Technically, the valuation of induction variables is analog to parameter passing in a purely functional language: each statement is considered as a function, binding and initializing one or more induction variables. Our induction variable characterization does not take the outcome of loop and test predicates into account.[1] Thus, we will consider a superset of the valid traces for the evaluation of induction variables. We propose two alternative algorithms for this evaluation. The result of both algorithms for each induction variable is a *binding function* mapping instances to the abstract memory locations they access. It is a *rational function* on the Cartesian product of two monoids and can be efficiently represented as a *rational transducer* [Ber79]. This binding function will give an *exact* result for valid traces.

Structure of the chapter. To focus on the core concepts and contributions, we introduce MOGUL, a domain-specific language with high-level constructs for traversing data structures addressed by induction variables in a *finitely presented monoid*. In a general-purpose (imperative or functional) language, our technique would require additional information about the shape of data structures, using dedicated annotations [HHN92, KS93, FM97] or shape analyses [GH96, SRW99]. Despite the generality of the control structures in MOGUL, as said before, binding functions give *exact* values for valid traces. This may be used to derive *alias* and *dependence* information of recursive programs with an unprecedented precision [Coh99, Col02, ACF03]. We will survey the current applications of instancewise analysis for recursive programs; the reader interested in more details (for loop nests or more general recursive programs) may refer to [Col02] for a pedagogical and synthetic presentation.

Section 2.1 describes the control structures and trace semantics of MOGUL. Section 2.2 defines the abstraction of runtime control points into instances. Section 2.3 extends induction variables to recursive control and data structures. Section 2.4 states the existence of rational binding functions from individual instances to individual data structure elements. Section 2.5 addresses the computation and representation of binding functions as rational transducers. We consider practical examples in Section 2.6. Section 2.7 gives two simple applications of instancewise analysis to program optimization and surveys the current state of the art.

2.1 Control Structures and Execution Traces

We consider a simplified notion of *execution trace* with emphasis on the identification of runtime control points. For our purpose, a *trace* is a sequence of symbols called *labels* that denotes a *complete* execution of a program. Each label registers either the *beginning* of a statement execution or its *completion*. A *trace prefix* is the trace of a partial execution, given by a prefix of a complete trace. In the remainder, we will consider trace prefixes instead of the intuitive notion of runtime control point.

Figure 2.1 presents our running example. It features a recursive call to the Toy function, nested in the body of a *for* loop, operating on an array A. Thus, there is no simple way to remove the recursion. In this paper, *we will construct a finite-state representation for the infinite set of trace prefixes of* Toy*, then compute an exact finite-state characterization of the elements of* A *accessed by a given trace prefix.*

2.1.1 Control Structures in the MOGUL Language

Figure 2.2 gives the MOGUL version of Toy. It abstracts the shape of array A through a monoid type Monoid_int. Induction variables i and k are bound to values in this monoid. Traversals of A are expressed through i, k and the monoid operation ·. Further explanations about MOGUL data structures and induction variables are deferred to Section 2.3. We present in Figure 2.3 a simplified version of the MOGUL syntax, focusing on the control structures.

This is a *C*-like syntax with some specific concepts. Italic non-terminals are defined elsewhere in the syntax: infinite binary wordelementary_statement covers the usual atomic statements, including assignments, input/output statements, void statements, etc.; infinite binary wordpredicate is a boolean expression; infinite binary

[1] This limitation can be overcome thanks to approximations and higher complexity algorithms. We will present our solutions in another paper.

```
int A[20];

void Toy(int n, int k) {
  if (k < n)
  {
    for (int i=k; i<=n;
         i+=2)
      {
        A[i] = A[i] + A[n-i];
        Toy(n, k+1);
      }
  }
  return
}

int main() {
  Toy(20, 0);
}
```

Figure 2.1: Program Toy in C

```
  structure Monoid_int A;

A function Toy(Monoid_int n, Monoid_int k) {
B   if (k < n)
C   {
D     for (Monoid_int i=k; i<=n;
d          i=i.2)
E     {
F         A[i] = A[i] + A[n-i] ;
G         Toy(n, k.1);
      }
    }

  }

H function main() {
I   Toy(20, 0);
  }
```

Figure 2.2: Program Toy in MoGuL

```
program       ::=function                                                              (S1)
                |function program                                                      (S2)

function      ::='function' infinite binary wordident '(' infinite binary wordformal_parameter_list ')'
                     block                                                             (S3)

block         ::=LABEL ':' '{' infinite binary wordinit_list statement_list '}'        (S4)
                |LABEL ':' '{' statement_list '}'                                      (S5)

statement_list::=ε                                                                     (S6)
                |LABEL ':' statement statement_list                                    (S7)

statement     ::=infinite binary wordelementary_statement ';'                          (S8)
                |infinite binary wordident '(' infinite binary wordactual_parameter_list ')' ';'   (S9)
                |'if' infinite binary wordpredicate block 'else' block                 (S10)
                |'for' '(' infinite binary wordinit_list ';' LABEL ':' infinite binary wordpredicate ';'
                     LABEL ':' infinite binary wordtranslation_list ')' block          (S11)
                |block                                                                 (S12)
```

Figure 2.3: Simplified MoGuL syntax (control structures)

wordinit_list contains a list of initializations for one or more loop variables, and infinite binary wordtranslation_list is the associated list of constant translations for those induction variables; `block` collects a sequence of statements, possibly defining some induction variables. Every executable part of a program is labeled, either by hand or by the parser.

2.1.2 Interprocedural Control Flow Graph

We start with an intuitive presentation of the trace semantics of a MoGuL program, using the Interprocedural Control Flow Graph (ICFG): an extended control flow graph [ASU86] with function call and return nodes. The ICFG associated to `Toy` is shown in Figure 2.4.

Each elementary statement, conditional and function call is a node of the ICFG. More specifically:

- one node is associated to each `block` entry;
- each `for` loop generates three nodes: initialization (entry), condition (termination), and iteration;
- a `return` node exists for each function call.

The iteration node follows the last node of the loop block and leads to the condition node. Given a function call c in the program source, there is an edge in the ICFG from the node associated to c to the corresponding function body. Moreover, there is an edge from the `return` node to the statement following the function call in the source program.

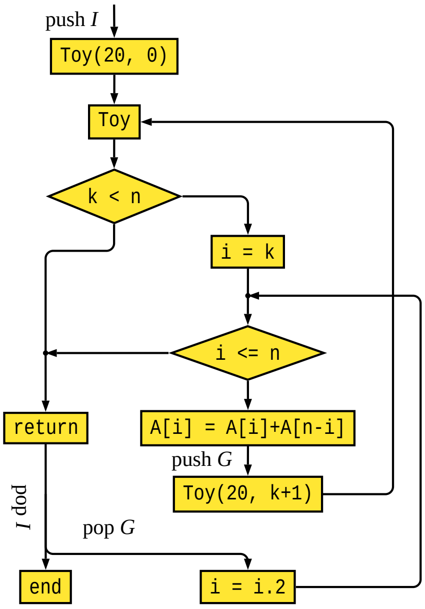


Figure 2.4: Interprocedural Control Flow Graph

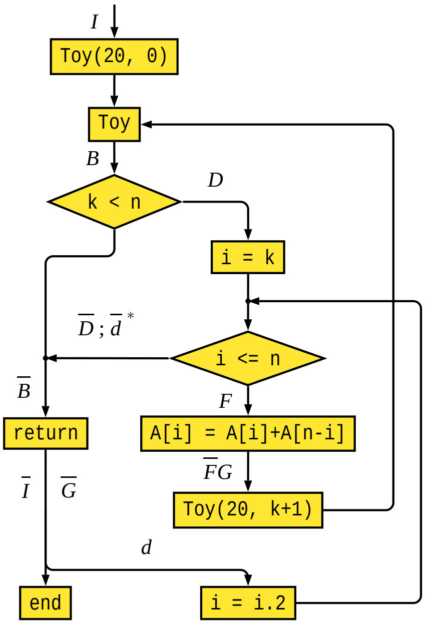


Figure 2.5: Simplified Pushdown Trace Automaton

To forbid impossible matchings of function calls and returns, i.e., to preserve context-sensitivity [NNH99], we provide the ICFG with a control stack [ASU86], see Figure 2.4. The result is the graph of a pushdown automaton. A complete trace is characterized as the word along a path from the initial node to the `end` node, the stack being empty at the latter node. We ignore the outcome of loop and test predicates, see Section 2.2. Consequently, some accepted paths correspond to valid execution traces, but others may still take wrong branches. Since we focus on a static scheme to name runtime control points, our trace semantics will make the same simplifying assumption and we will consider a superset of the valid traces.

2.1.3 The Pushdown Trace Automaton

Although MOGUL uses a *C* syntax, the instancewise framework in Section 2.2 considers each statement as a call to a function implementing elementary operations, conditional branches and iteration (as in a purely functional language). We extend the control stack of the ICFG to take these implicit calls into account. The stack alphabet now holds every statement label. Moreover, each statement is provided an additional label to separate the implicit function call from the implicit return. If ℓ a label, ℓ corresponds to the beginning of the execution of a statement, and $\overline{\ell}$ indicates its completion. Regarding the control stack, ℓ pushes ℓ while $\overline{\ell}$ pops ℓ. An additional state, called *return state*, is associated to the completion of each statement. The result is called the *pushdown trace automaton* and the recognized words are the *execution traces*.

When all states are considered final, the automaton recognizes all *trace prefixes*. It also recognizes prefixes of *non-terminating* traces in case the program loops indefinitely. We thus exclude non-terminating programs in the following.

Figure 2.6 presents the trace pushdown automaton of the `Toy` program. We exhibit here a prefix of a valid trace:
$IABCD\delta EF\overline{F}GABCD\delta EF\overline{F}GAB\overline{BAGE}\delta d\delta EF$

For clarity of exposure and figures relative to the running example and without loss of precision, we use a simplified representation of the trace pushdown automaton in Figure 2.5: it omits return states, except for `Toy` calls, and states associated to `block` statements and to loop predicates. Now, the previous trace prefix reduces to: $IBDF\overline{F}GBDF\overline{F}GB\overline{BG}dF$. We will use this simplified representation of traces in the following.

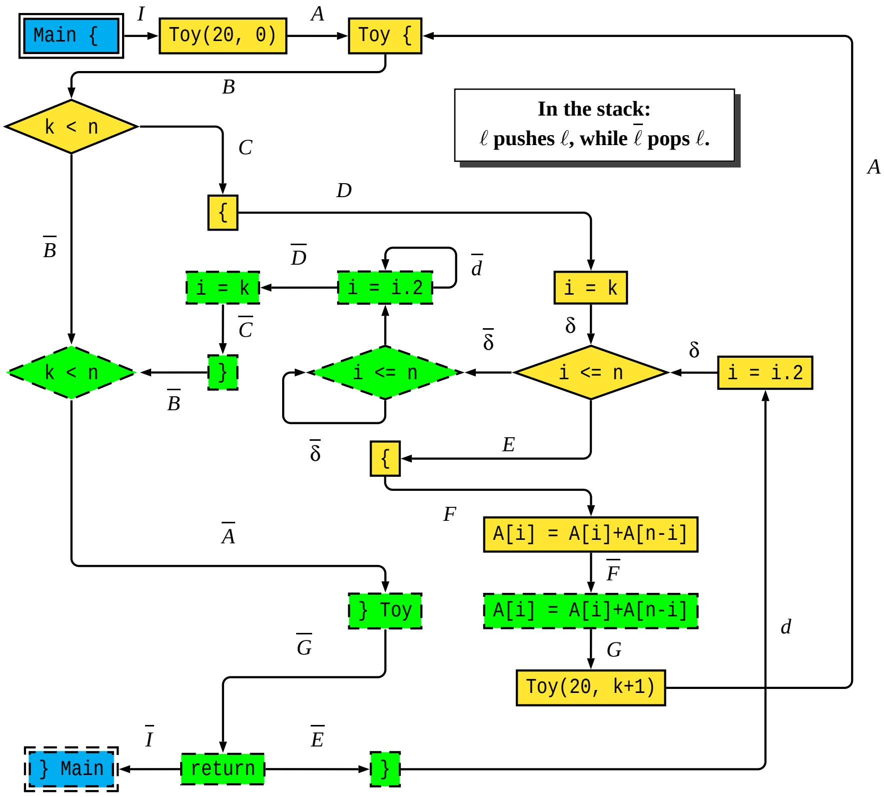


Figure 2.6: Pushdown Trace Automaton

2.1.4 The Trace Grammar

After the intuitive presentation above, this section gives a formal definition of traces. There is one context-free trace grammar G_P per program P.

1. For each call to a function `id`, i.e., each derivation of production ($S9$), there is a production schema

$$C_{\mathtt{id}} \quad ::= \quad \mathtt{Label}\ B_{\mathtt{id}}\ \overline{\mathtt{Label}} \tag{2.1}$$

   where C_{id} and B_{id} are the respective non-terminals of the function call and body. `Label` is the terminal label of the call to function `id`, and $\overline{\mathtt{Label}}$ marks the end of the statement, here a `return` statement.

2. For each loop statement s, i.e., each derivation of production ($S11$), there are four production schemas

$$L_s \quad ::= \quad \varepsilon \quad | \quad \mathtt{Label}_{\mathtt{e}}\ \mathtt{Label}_{\mathtt{p}}\ B_s\ O_s\ \overline{\mathtt{Label}_{\mathtt{p}}}\ \overline{\mathtt{Label}_{\mathtt{e}}} \tag{2.2}$$

$$O_s \quad ::= \quad \varepsilon \quad | \quad \mathtt{Label}_{\mathtt{i}}\ \mathtt{Label}_{\mathtt{p}}\ B_s\ O_s\ \overline{\mathtt{Label}_{\mathtt{p}}}\ \overline{\mathtt{Label}_{\mathtt{i}}} \tag{2.3}$$

   where the three non-terminals L_s, O_s and B_s correspond to the loop entry, iteration and body, respectively. $\mathtt{Label}_{\mathtt{e}}$, $\mathtt{Label}_{\mathtt{p}}$ and $\mathtt{Label}_{\mathtt{i}}$ are terminals, they are the labels of the loop entry, predicate and iteration, respectively.

3. For each conditional s, i.e., each derivation of production ($S10$), there are two productions schemas

$$I_s \quad ::= \quad \mathtt{Label}\ T_s\ \overline{\mathtt{Label}} \quad | \quad \mathtt{Label}\ F_s\ \overline{\mathtt{Label}} \tag{2.4}$$

   where the three non-terminals I_s, T_s and F_s correspond to the conditional, `then` branch and `else` branch, respectively. `Label` is the terminal label of the conditional.

4. For each block s, i.e., each derivation of productions ($S4$) or ($S5$), there is a production schema

$$B_s \quad ::= \quad \mathtt{Label}\ S_1\ \ldots\ S_n\ \overline{\mathtt{Label}} \tag{2.5}$$

where the non-terminal B_s corresponds to the block and non-terminals $S_1, \dots, S_n$ correspond to each statement in the block. `Label` is the terminal label of B_s.

5. For each elementary statement s, there is a production schema

$$S_s \quad ::= \quad \texttt{Label}\,\overline{\texttt{Label}} \tag{2.6}$$

where `Label` is the terminal label of statement s.

The axiom of the trace grammar is the non-terminal associated with the block of the `main` function.

Definition 1 (Trace Language) *The set of traces of a program P — called the* trace language *of P — is the set of terminal sentences of G_P.*

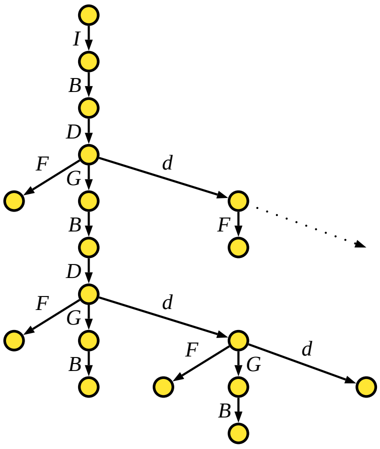
For a given execution trace t, runtime control points are sequentially ordered according to the appearance of statement labels in t.

Definition 2 (Sequential Order) *The* sequential order $<_{seq}$ *is the strict prefix order of the trace prefixes. It is a total order for a given execution trace.*

Calling L_{ab} the alphabet of labels, the *trace language* recognized by G_P is a context-free (a.k.a. algebraic) subset of the free monoid L_{ab}^*, and ε denotes its empty word. Clearly, the trace language fits the intuition about program execution and the previous presentation in terms of the interprocedural control flow graph: the pushdown trace automaton recognizes the trace language.

Grammar G_P generates many terminal sentences that are possible execution sequences for P. These sentences depend on choices between productions (2.1) to (2.6). In a real execution of P, these choices are dictated by the outcome of loop and test predicates, which our grammar does not take into account. It is customary to say that predicates are not interpreted (in the model theory sense), or that P is a *program schema* [Man74]. We are free to select which predicates and operations should be interpreted: e.g., the polytope model interprets every loop bound and array subscript in number theory [PD96]. In this paper, we will interpret address computations in the theory of finitely-presented monoids; everything else will remain uninterpreted.

Eventually, a runtime execution may be represented in the shape of an *activation tree* [ASU86]: the sequential execution flow corresponds to the depth-first traversal of the activation tree. This representation is used in the formal definition of instances. Figure 2.7 shows an activation tree for `Toy`. We label each arc according to the target node statement. The trace is obtained while reading the word along the depth-first traversal: each downward step produces the arc label, and each upward step produces the associated overlined label.



`Trace prefix:` $IBDF\overline{F}GBDF\overline{F}GB\overline{BG}dF\overline{F}GB\overline{BG}d\overline{ddDBG}dF$

Figure 2.7: Activation tree

2.2 The Instancewise Model

This section is dedicated to the first part of our framework: the abstraction of *trace prefixes* into *control words*, the formal representation of *instances*. The control word abstraction characterizes an infinite set of trace prefixes in a tractable, finite-state representation. We present the properties of control words from several points of view: pushdown trace automata, trace prefixes, activation trees, and MoGuL grammar. This last insight introduces a control words grammar that generates a superset of control words. We then investigate the conditions realizing the equivalence of the language generated by the control words grammar and the set of control words. This section ends with the description of the control word language in the form of a finite-state automaton, a counterpart of the pushdown trace automaton. Finally, we expose one of the main results of this work, justifying the introduction of control words as the basis for instancewise analysis.

2.2.1 From the Pushdown Trace Automaton to Control Words

The pushdown trace automaton will help us prove an important property of control words.

Definition 3 (Stack Word Language) *The stack word language of a pushdown automaton $\mathcal{A}$ is the set of stack words u such that there exists a state q in $\mathcal{A}$ for which the configuration (q,u) is both accessible and co-accessible — there is an accepting path traversing q with stack word u.*

Definition 4 (Control Word) *The stack word language of the pushdown trace automaton is called the* control word language. *A* control word *is the sequence of labels of all statements that have begun their execution but not yet completed it. Any trace prefix has a corresponding control word.*

Since the stack word language of a pushdown automaton is rational [RS97a], we have:

Theorem 1 *The language of control words is rational.*

The activation tree is a convenient representation of control words. When the label of node n is at the top of the control stack, the control word is the sequence of labels along the *branch* of n in the activation tree, i.e., the path from the root to node n [ASU86]. Conversely, a word labeling a branch of the activation tree is a control word. For example, $IBDdF$ is the control word of trace prefix $IBDF\overline{F}GBDF\overline{F}GB\overline{B}\overline{G}dF\overline{F}GB\overline{B}\overline{G}d\overline{d}\overline{d}\overline{D}\overline{B}\overline{G}dF$ in Figure 2.7.

2.2.2 From Traces to Control Words

The trace language is a Dyck language [Ber79], i.e., a hierarchical parenthesis language. The restricted Dyck congruence over L^*_{ab} is the congruence generated by $\ell\overline{\ell} \equiv \varepsilon$, for all $\ell \in L_{\mathrm{ab}}$.[2] This definition induces a rewriting rule over L^*_{ab}, obviously confluent. This rule is the direct transposition of the control stack behavior. Applying it to any trace prefix p we can associate a minimal word w.

Lemma 1 *The control word w associated to the trace prefix p is the shortest element in the class of p modulo the restricted Dyck congruence.*

Definition 5 (Slimming Function) *The* slimming function *maps each trace prefix to its associated control word.*

Theorem 2 *The set of control words is the quotient set of trace prefixes modulo the restricted Dyck congruence, and the slimming function is the canonical projection of trace prefixes over control words.*

From now on, the restricted Dyck congruence will be called the *slimming congruence*. The following table illustrates the effect of the slimming function on a few trace prefixes.

Trace prefix	$IBDF\overline{F}GBDF$
Control word	$IBD\quad GBDF$
Trace prefix	$IBDF\overline{F}GBDF\overline{F}GB\overline{B}\overline{G}dF\overline{F}G$
Control word	$IBD\quad GBD\qquad d\quad G$
Trace prefix	$IBDF\overline{F}GBDF\overline{F}GB\overline{B}\overline{G}dF\overline{F}GB\overline{B}\overline{G}d\overline{d}\overline{d}\overline{D}\overline{B}\overline{G}dF$
Control word	$IBD\qquad dF$

The slimming function extends Harrison's NET function, and control words are very similar to his *procedure strings* [Har89]. Harrison introduced these concepts for a statementwise analysis with dynamic partitioning.

[2] The *restricted* qualifier means that only $\ell\overline{\ell}$ couples are considered, $\overline{\ell}\ell$ being a nonsensical sub-word for the trace grammar.

2.2.3 From the Trace Grammar to Control Words

We may also derive a *control words grammar* from the trace grammar. This grammar significantly differs from the trace grammar in three ways.

1. Control words contain no overlined labels.

   The control stack ignores overlined labels.

2. Each non-terminal is provided an empty production.

   A control word is associated to each trace prefix.

3. If the right-hand side of a production consists of multiple non-terminals, it is replaced by an individual production for each non-terminal.

   Only the last statement of an uncompleted sequence remains in the control stack, i.e., in the control word.

Under these considerations, the productions for the control words grammar are the following, with the same notations and comments as the trace grammar.

1. For each function call `id`, i.e., each derivation of production ($S9$), there are two productions

$$C_{\texttt{id}} \quad ::= \quad \texttt{Label}\, B_{\texttt{id}} \quad | \quad \varepsilon$$

2. For each loop statement s, i.e., each derivation of production ($S11$), there are six productions

$$\begin{array}{rcl} L_s & ::= & \texttt{Label}_\texttt{e}\,\texttt{Label}_\texttt{p}\, B_s \quad | \quad \texttt{Label}_\texttt{e}\, O_s \quad | \quad \varepsilon \\ O_s & ::= & \texttt{Label}_\texttt{i}\,\texttt{Label}_\texttt{p}\, B_s \quad | \quad \texttt{Label}_\texttt{i}\, O_s \quad | \quad \varepsilon \end{array}$$

3. For each conditional s, i.e., each derivation of production ($S10$), there are three productions

$$I_s \quad ::= \quad \texttt{Label}\, T_s \quad | \quad \texttt{Label}\, F_s \quad | \quad \varepsilon$$

4. For each block s enclosing n statements, i.e., each derivation of ($S4$) or ($S5$), there are $n+1$ productions

$$B_s \quad ::= \quad \texttt{Label}\, S_1 \quad | \quad \cdots \quad | \quad \texttt{Label}\, S_n \quad | \quad \varepsilon$$

5. For each elementary statement s,

$$S_s \quad ::= \quad \texttt{Label} \quad | \quad \varepsilon$$

The axiom of this grammar is the block of the `main` function.

The control words grammar grammar above is right linear,[3] hence its generated language is rational.

Lemma 2 *The language of control words is a subset of the language generated by the control words grammar.*

The proof comes from the three above observations that translate the effect of the slimming function. For each trace grammar derivation, we associate a corresponding derivation of the control words grammar. The control words grammar generates any stack word corresponding to a path — accepting or not — in the pushdown trace automaton.

The next section will show that the control words grammar only generates control words, assuming the trace grammar satisfies a termination criterion.

2.2.4 Control Words and Program Termination

Assuming *any incomplete execution can be completed until the termination of the program*, stack words corresponding to a path of the pushdown automaton are all stack words of trace prefixes, i.e., control words.

Conversely, if a partial execution has entered a step where the last opened statement can never be completed, a recursive cycle in the trace derivation cannot be avoided.

[3] At most one non-terminal in the right-hand side, and non-terminals are right factors.

Example

Consider the following trace grammar:

$$\begin{array}{lclclcl} S & \rightarrow & aAb\overline{ba} & \qquad & B & \rightarrow & fC\overline{f} \\ A & \rightarrow & cB\overline{c} & & C & \rightarrow & gB\overline{g} \\ A & \rightarrow & de\overline{ed} & & & & \end{array}$$

a labels the body of function `main` and b labels an elementary statement. A is a non-terminal for a conditional test; function B is called in the `then` branch, while elementary statement s is executed in the `else` one. Function B calls function C and conversely. Thus, the `then` branch may never terminate. The corresponding control words grammar is:

$$\begin{array}{lclclcl} S & \rightarrow & aA & \quad & A & \rightarrow & \varepsilon \\ S & \rightarrow & ab & & B & \rightarrow & fC \\ S & \rightarrow & \varepsilon & & B & \rightarrow & \varepsilon \\ A & \rightarrow & cB & & C & \rightarrow & gB \\ A & \rightarrow & de & & C & \rightarrow & \varepsilon \end{array}$$

This grammar generates ac, thanks to the derivation

$$S \rightarrow aA;\ A \rightarrow cB;\ B \rightarrow \varepsilon.$$

However, no trace prefix can be generated by the trace grammar for which the control word is ac, hence ac is not a control word. To avoid this, we need a criterion that forbids recursive trap cycles. This criterion is defined through the structure of the trace grammar; we refer to the definition of a *reduced grammar* [TS85].

Definition 6 (Reduced Grammar) *A* reduced *grammar is a context-free grammar such that:*

1. *there is no $A \rightarrow A$ rule;*
2. *any grammar symbol occurs in some sentential form (a sentential form is any derivative from the axiom);*
3. *any non-terminal produces some part of a terminal sentence.*

The third rule is the criterion we are looking for: a non-terminal which produces some part of a terminal sentence is said *active*. The control words grammar of the program must have only active non-terminals; it is called an *unlooping* grammar. In the previous example, B and C are *not active*.

Termination criterion for the trace grammar

Starting from a set of non-terminals N, we recall an inductive algorithm that determines the set of active non-terminals $N' \subseteq N$; if $N = N'$, the grammar is unlooping [TS85]. The initial set N'_1 contains active non-terminals that immediately produce a part of a terminal sentence; Φ denotes the set of grammar rules, T is the set of terminals, and m is the cardinal of N.

Algorithm 1

```
$N'_1 \leftarrow \{A \mid A \rightarrow \alpha \in \Phi \wedge \alpha \in T^*\}$
For $k = 2, 3, \ldots, m$
    $N'_k \leftarrow N'_{k-1} \bigcup \{A \mid A \rightarrow \alpha \in \Phi \wedge \alpha \in (T \bigcup N'_{k-1})^+\}$
    If $N'_k = N'_{k-1} \vee k = m$
    Then $N' \leftarrow N'_k$
```

Applied to our example where $N = \{S, A, B, C\}$:

$$N'_1 = \{A\};\ N'_2 = \{A, S\};\ N'_3 = N'_2;\ N' = \{A, S\};\ N \neq N'.$$

Thanks to Lemma 2, we may state a necessary and sufficient condition for the control words grammar to only generate control words.

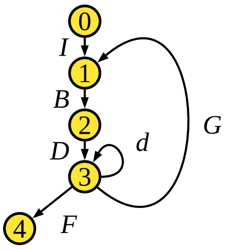
Theorem 3 *Let P be a program given by its trace grammar G_P, and let G'_P be the associated control words grammar. The control words language of P is generated by G'_P if and only if Algorithm 1 concludes that G_P is unlooping.*

2.2.5 The Control Automaton

We now assume the program satisfies Theorem 3.

It is easy to build a finite-state automaton accepting the language generated by the right-linear control words grammar, i.e., a finite-state automaton recognizing the language of control words. We call the latter the *control automaton*.

Figure 2.8 shows the control automaton for `Toy`; the control word language is $I + IB + IBD(d + GBD)^*(\varepsilon + F + G + GB)$.

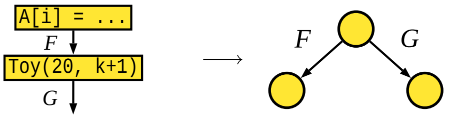


All states are final.

A few control words:

IBDdF,
IBDGBDF,
IBDGBDdG.

Figure 2.8: Example Control Automaton



Each statement in a sequence is linked to the enclosing block.

Figure 2.9: Construction of the Control Automaton

The transformation from traces to control words is a systematic procedure. A similar transformation exists from the pushdown trace automaton to the control automaton; this is important for the design of efficient instancewise analysis algorithms (see Section 2.4).

- In the pushdown trace automaton, a sequence of successive statements is a chain of arcs, while, in the control automaton, each of these statement is linked by an edge from the common enclosing block, see Figure 2.9. Thus, the control automaton makes no distinction between the sequence and the conditional.
- As in the pushdown automaton for trace prefixes, all states are final.
- Since a `return` statement closes the corresponding function call and deletes every label relative to it in the control word, `return` nodes are not needed anymore.

2.2.6 Instances and Control Words

Consider any trace t of a MOGUL program and any trace prefix p of t. The slimming function returns a unique control word. Conversely, it is easy to see that a given control word may be the abstraction of many trace prefixes, possibly an infinity. E.g., consider two trace prefixes differing only by the sub-trace of a completed conditional statement:[4] their control words are the same.

This section will prove that, during any execution of a MOGUL program, the stack that registers the control word at runtime cannot register twice the same control word (i.e., for two distinct trace prefixes). In others words, control words characterize runtime control points in a more compact way than trace prefixes. For the demonstration, we introduce a strict order over control words.

Definition 7 (Lexicographic Order) *We first define the partial* textual order $<_{lab}$ *over labels. Given* s_1 *and* s_2 *two labels in* L_{ab}, $s_1 <_{lab} s_2$ *if and only if*

- *there is a production generated by (2.5) in the trace grammar, such as* s_1 *is the label of* S_i *and* s_2 *is the label of* S_j, *with* $1 \leq i < j \leq n$;
- *or there is a production generated by (2.2) or (2.3) such as* s_1 *is the label of* B_s *and* s_2 *is the label of* O_s.

[4] I.e., after both branches have been completed, the first sub-trace denoting the `then` branch and the other the `else` one.

We denote by $<_{lex}$ the strict lexicographic order over control words induced by $<_{lab}$.

In other words, $<_{\mathrm{lab}}$ is the textual order of appearance of statements within blocks, considering the loop iteration statement as textually ordered after the loop body.

Lemma 3 *The sequential order $<_{seq}$ over prefix traces is compatible with the slimming congruence. The lexicographic order $<_{lex}$ is the quotient order induced by $<_{seq}$ through the slimming congruence.*

The proof takes two steps. First of all, let t be a trace and T its activation tree. The set of all paths in T is ordered by a strict lexicographic order, $<_T$, isomorphic to $<_{\mathrm{lex}}$.

Then, let α be the function mapping any path in T to the last label of the path word (accurately speaking of the control word labeling this path). Given a trace prefix p and the $<_T$ ordered sequence $\{b_0 = \varepsilon, b_1, ..., b_n\}$ of all paths in T, the (partial) depth-first traversal of T until p yields the following word:

$$\mathrm{dft}(p) \triangleq \alpha(b_0)\alpha(b_1)...\alpha(b_q),$$

where b_q is the branch of p, $q \leq n$. Now, the definition of $\mathrm{dft}(p)$ is precisely p.

Let p_q and p_r be two prefixes of t, p_q being a prefix of p_r itself, and write

$$p_q = \alpha(b_0)\alpha(b_1)...\alpha(b_q), p_r = \alpha(b_0)\alpha(b_1)...\alpha(b_r).$$

We have the following: $p_q <_{\mathrm{seq}} p_r \iff b_q <_T b_r$. Together with the first step, $p_q <_{\mathrm{seq}} p_r \iff b_q <_{\mathrm{lex}} b_r$.

We now come to the formal definition of instances.

Definition 8 (Instance) *For a* MOGUL *program, an* instance *is a class of trace prefixes modulo the slimming congruence.*

It is fundamental to notice that, in this definition, instances do not depend on any particular execution.

From Lemma 3 and Theorem 2 (the slimming function is the canonical projection of trace prefixes to control words), we may state the two main properties of control words.

Theorem 4 *Given one execution trace of a* MOGUL *program, trace prefixes are in bijection with control words.*

Theorem 5 *For a given* MOGUL *program, instances are in bijection with control words.*

Theorem 4 ensures the correspondence between runtime control points and control words. Theorem 5 is just a rewording of Theorem 2, it states the meaning of control words across multiple executions of a program.

In the following, we will refer to instances or control words interchangeably, without naming a particular trace prefix representative.

2.3 Data Structure Model and Induction Variables

This section and the following ones apply instancewise analysis to the *exact characterization of memory locations* accessed by a MOGUL program. For decidability reasons, we will only consider a restricted class of data structures and addressing schemes:

- data structures do not support destructive updates (deletion of nodes and non-leaf insertions);[5]
- addressing data-structures is done through so called induction variables whose only authorized operations are the initialization to a constant and the associative operation of a monoid.

These restrictions are reminiscent of *purely functional data structures* [Oka96].

In this context, we will show that the value of an induction variable at some runtime control point — or the memory location accessed at this point — only depends on the instance. Exact characterization of induction variables will be possible at compile-time by means of so-called *binding functions* from control words to abstract memory locations (monoid elements), independently of the execution.

[5] Leaf insertions are harmless if data-structures are implicitly expanded when accessed.

2.3.1 Data Model

To simplify the formalism and exposition, MOGUL data structures with side-effects must be *global*. This is not really an issue since any local structure may be "expanded" along the activation tree (e.g., several local lists may be seen as a global stack of lists).

A *finitely-generated monoid* $M = (G, \equiv)$ is specified by a *finite* list of *generators* G and a *congruence* $\equiv$ given by a *finite* list of equations over words in G^*. Elements of M are equivalence classes of words in G^* modulo $\equiv$. When the congruence is empty, M is a *free monoid*. The operation of M is the quotient of the concatenation on the free monoid G^* modulo $\equiv$; it is an associative operation denoted by $\cdot$ with neutral element ε_m.

Definition 9 (Abstract Location) *A data structure is a pair of a* data structure name *and a finitely-generated monoid* $M = (G, \equiv)$*. An abstract memory location in this data structure is an element of the monoid. It is represented by an* address word *in* G^**. By definition, two congruent address words represent the same memory location.*

Typical examples are the n-ary tree — the free monoid with n generators (with an empty congruence) — and the n-dimensional array — the free commutative monoid $\mathbb{Z}^n$ (with vector commutation and inversion).

Below are listed some practical examples of monoid-based data structures.

Free monoid.
$G = \{\texttt{right}, \texttt{left}\}$, $\equiv$ is the identity relation, $\cdot$ is the concatenation: monoid elements address a binary tree.

Free group.
$G = \{\texttt{right}, \texttt{left}, \texttt{right}^{-1}, \texttt{left}^{-1}\}$, $\equiv$ is the inversion of `left` and `right` (without commutation): Cayley graphs [ECH$^+$92, GMS95].

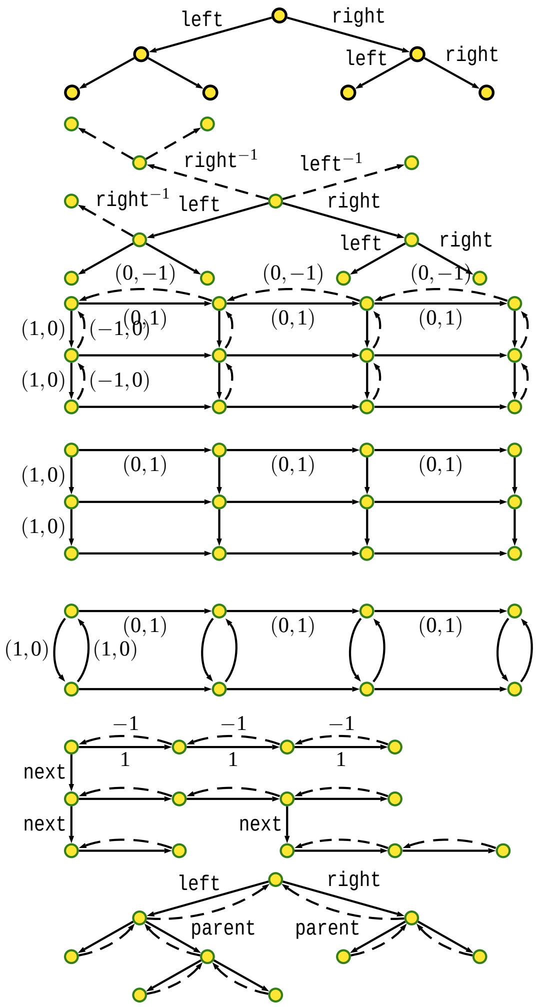
Free commutative group.
$G = \{(0,1), (1,0), (0,-1), (-1,0)\}$, $\equiv$ is the vector inversion and commutation, $\cdot$ is vector addition: a two-dimensional array.

Free commutative monoid.
$G = \{(0,1), (1,0)\}$, $\equiv$ is vector commutation: a two-dimensional grid.

Commutative monoid.
$G = \{(0,1), (1,0)\}$, $\equiv$ is vector commutation and $(0,1) \cdot (0,1) \equiv \varepsilon_m$: a two-dimensional grid folded on the torus $\mathbb{Z} \times \frac{\mathbb{Z}}{2\mathbb{Z}}$.

Free partially-commutative monoid.
$G = \{\texttt{next}, 1, -1\}$, $\equiv$ is the inversion and commutation of 1: nested trees, lists and arrays.

Monoid with right-inverse.
$G = \{\texttt{right}, \texttt{left}, \texttt{parent}\}$, $\texttt{right} \cdot \texttt{parent} \equiv \varepsilon_m$, $\texttt{left} \cdot \texttt{parent} \equiv \varepsilon_m$: a tree with backward edges.



2.3.2 Induction Variables

Traditionally, induction variables are scalar variables within loop nests with a tight relationship with the surrounding loop counters [ASU86, GSW95]. This relationship, deduced from the regularity of the induction variable updates, is a critical information for many analyses (dependence, array region, array bound checking) and optimizations (strength-reduction, loop transformations, hoisting).

A *basic linear induction variable* `x` is assigned (once or more) in a loop, each assignment being in the form `x =` c or `x = x +` c, where c is a constant known at compile-time. More generally, a variable `x` is called a *linear induction variable* if on every iteration of the surrounding loop, `x` is added a constant value. This is the case when assignments to `x` in the cycle are in the basic form or in the form `x = y +` c, `y` being another induction variable. The value of `x` may then be computed as an affine function of the surrounding loop counters.

MOGUL extensions are twofold:

- induction variables are not restricted to arrays but handle all monoid-based data structures;
- both loops and recursive function calls are considered.

As a consequence, induction variables represent abstract addresses in data structures, and the basic operation over induction variables becomes the monoid operation.

Definition 10 (Induction Variable) *A variable* `x` *is an induction variable if and only if the three following conditions are satisfied:*

a. `x` *is defined at a* `block` *entry, a* `for` *loop initialization, or* `x` *is a formal parameter;*

b. `x` *is constant in the* `block`*, the* `for` *loop or the function where it has been defined;*

c. *the definition of* `x` *(according to* **a***) is in one of the forms:*

1. `x =` c*, and* c *is a constant known at compile-time,*
2. `x = y ·` c*, and* `y` *is an induction variable, possibly equal to* `x`*.*

A MOGUL induction variable can be used in different address expressions which reference *distinct* data structures, provided these structures are defined over the same monoid. This separation between data structure and shape follows the approach of the declarative language 8$_{1/2}$ [GMS95]. It is a convenient way to expose more semantics to the static analyzer, compared with C pointers or variables of product types in ML.

Eventually, the MOGUL syntax is designed such that *every variable of a monoid type is an induction variable*, other variables being ignored. The only valid definitions and operations on MOGUL variables are those satisfying Definition 10. For any monoid shape, data structure accesses follow the *C* array syntax: `D[a]` denotes element with address a of structure `D`, where a is in the form `x` or `x ·` c, `x` an induction variable and c a constant.

If `A` is an array (i.e., `A` is addressed in a free commutative group), the affine subscript `A[i+2j-1]` is not a valid MOGUL syntax. This is not a real limitation, however, since affine subscripts may be replaced by new induction variables defined when necessary while `i` or `j` are defined. As an illustration, let `k` be the induction variable equal to `i+2j-1`, the subscript in the reference above. We have to build, through a backward motion, static chains of induction variables from the program start point to the considered reference. Suppose the last modification of the subscript before the considered program point is given by the statement `j= h` denoted by s, where `h` is another induction variable. We have to define a new induction variable `g = i+2h-1`, living before this statement, and to consider that s initializes `k` through an additional assignment `k= g`. This work has to be done recursively for all paths in the control flow graph until reaching the start point.

2.4 Binding Functions

In MOGUL, the computations on two induction variables in two distinct monoids are completely separate. Thus, without loss of generality, we suppose that all induction variables belong to a single monoid M_{loc}, with operation $\cdot$ and neutral element ε_m, called the *data structure monoid*.

2.4.1 From Instances to Memory Locations

In a purely functional language, function application is the only way to define a variable. In MOGUL, every statement is handled that way; the scope of a variable is restricted to the statement at the beginning of which it has been declared, and an induction variable is constant in its scope.

Since overloading of variable names occurs at the beginning of each statement, the value of an induction variable depends on the runtime control point of interest. Let x be an induction variable, we define the *binding for* x as the pair (p, v_p), where p is a trace prefix and v_p the value of x after executing p.

Consider two trace prefixes p_1 and p_2 representative of the same instance. The previous rules guarantee that all induction variables living right after p_1 (resp. p_2) have been defined in statements not closed yet. Now, the respective sequences of non-closed statements for p_1 and p_2 are identical and equal to the control word of p_1 and p_2. Thus the bindings of x for p_1 and p_2 are equal. In others words, the function that binds the trace prefix to the value of x is compatible with the slimming congruence.

Theorem 6 *Given an induction variable* x *in a* MOGUL *program, the function mapping a trace prefix p to the value of* x *only depends on the instance associated to p, i.e., on the control word.*

In other words, given an execution trace, the bindings at any trace prefix are identified by the control word (i.e., the instance).

Definition 11 (Binding Function) *A* binding for x *is a couple* (w, v)*, where w is a control word and v the value of* x *at the instance w.*

$\Lambda_{\texttt{x}}$ *denotes the* binding function for x*, mapping control words to the corresponding value of* x*.*

2.4.2 Bilabels

We now describe the mathematical framework to compute binding functions.

Definition 12 (Bilabel) *A* bilabel *is a pair in the set* $L_{ab}^{*} \times M_{loc}$*. The first part of the pair is called the* input label, *the second one is called the* output label.

$B = L_{\mathrm{ab}}^{*} \times M_{\mathrm{loc}}$ denotes the set of bilabels. From the *direct product* of the control word free monoid L_{ab}^{*} and the data monoid M_{loc}, B is provided with a monoid structure: its operation $\bullet$ is defined componentwise on L_{ab}^{*} and M_{loc},

$$(\alpha|a) \bullet (\beta|b) \overset{def}{=} (\alpha\beta|a \cdot b). \tag{2.7}$$

A binding for an induction variable is a bilabel. Every statement updates the binding of induction variables according to their definitions and scope rules, the corresponding equations will be studied in Section 2.4.3.

Definition 13 *The set of* rational subsets *of a monoid M is the least set that contains the finite subsets of M, closed by union, product and the star operation [Ber79].*

A rational relation *over two monoids M and* M' *is a rational subset of the monoid* $M \times M'$*.*

We focus on the family B_{rat} of rational subsets of B.

Definition 14 *A semiring is a monoid for two binary operations, the "addition"* $+$*, which is commutative, and the "product"* $\times$*, distributive over* $+$*; the neutral element for* $+$ *is the zero for* $\times$*.*

The powerset of a monoid M is a semiring for union and the operation of M [Ber79]. The set of rational subsets of M is a sub-semiring of the latter [Ber79]; it can be expressed through the set of rational expressions in M. Thus B_{rat} is a semiring.

We overload $\bullet$ to denote the product operation in B_{rat}; $\emptyset$ is the zero element (the empty set of bilabels); and the neutral element for $\bullet$ is $\mathcal{E} = \{(\varepsilon, \varepsilon_m)\}$. From now on, we identify B_{rat} with the set of rational expressions in M, and we also identify a singleton with the bilabel inside it: $\{(s|c)\}$ may be written $(s|c)$.

2.4.3 Building Recurrence Equations

To compute a finite representation of the binding function for each induction variable, we show that the bindings can be expressed as a finite number of rational sets. First of all, bindings can be grouped according to the last executed statement, i.e., the last label of the control word. Next, we build a system of equations in which unknowns are sets of bindings for induction variable x at state n of the control automaton. Given $\mathcal{A}_n$ the control automaton modified so that n is the unique final state, let $\mathcal{L}_n$ be the language recognized by $\mathcal{A}_n$. The *binding function for* x *at state n*, $\Lambda_{\texttt{x}}^{n}$, is the binding function for x restricted to $\mathcal{L}_n$. We also introduce a new induction variable z, *constant and equal to* ε_m.

The system of equations is a direct translation of the semantics of induction variable definitions; it follows the syntax of a MOGUL program P; we illustrate each rule on the running example.

1. At the initial state 0 and for any induction variable `x`,

$$\Lambda^0_{\mathtt{x}} = \mathcal{E} \tag{2.8}$$

E.g., the `Toy` program involves three induction variables, the loop counter `i` and the formal parameters `k` and `n`. We will not consider `n` since it does not subscript any data structure. The output monoid is $\mathbb{Z}$, its neutral element ε_m is 0.

$$\Lambda^0_{\mathtt{k}} = \Lambda^0_{\mathtt{i}} = (\varepsilon|0).$$

2. $\Lambda^n_{\mathtt{z}}$ denotes the set defined by

$$\Lambda^n_{\mathtt{z}} = \bigcup_{w \in \mathcal{L}_n} (w|\varepsilon_m). \tag{2.9}$$

$\Lambda^n_{\mathtt{z}}$ is the binding function for the new induction variable `z` restricted to $\mathcal{L}_n$; it is constant and equal to ε_m.

For each statement s defining an induction variable `x` to $c_{s\mathtt{x}}$ (case **c**.1 of Definition 10), and calling d and a the respective departure and arrival states of s in the control automaton,

$$\Lambda^a_{\mathtt{x}} \supseteq \Lambda^d_{\mathtt{z}} \bullet (s|c_{s\mathtt{x}}). \tag{2.10}$$

Since $\Lambda^d_z \bullet (s|c_{s\mathtt{x}}) = \bigcup_{w \in \mathcal{L}_d} (ws|c_{s\mathtt{x}})$, (2.10) means: if $w \in \mathcal{L}_d$ is a control word, ws is also a control word and its binding for `x` is $(ws|c_{s\mathtt{x}})$.

The control automaton automaton of `Toy` has 5 states. For the case **c**.1 of Definition 10,

$$\text{statement } I: \qquad \mathtt{k\ =\ 0}, \tag{2.11}$$

and (2.10) yields

$$\Lambda^1_{\mathtt{k}} \quad \supseteq \quad \Lambda^0_{\mathtt{z}} \bullet (I|0).$$

3. For each statement s defining an induction variable `x` to $\mathtt{y} \cdot c$ (case **c**.2 of Definition 10), and d and a the respective departure and arrival states of s,

$$\Lambda^a_{\mathtt{x}} \quad \supseteq \quad \Lambda^d_{\mathtt{x}} \bullet (s|c_{s\mathtt{x}}). \tag{2.12}$$

To complete the system, we add for every induction variable `x` unchanged by s a set of equations in the form (2.12), where $c_{s\mathtt{x}} = \varepsilon_m$.

E.g., for case **c**.2 of Definition 10,

$$\text{statement } G: \qquad \mathtt{k\ =\ k} \cdot 1 \tag{2.13}$$

$$\text{statement } d: \qquad \mathtt{i\ =\ i} \cdot 2 \tag{2.14}$$

$$\text{statement } D: \qquad \mathtt{i\ =\ k} \tag{2.15}$$

and (2.12) yields

$$\begin{array}{ll}
\Lambda^1_{\mathtt{i}} \supseteq \Lambda^3_{\mathtt{i}} \bullet (G|0) & \Lambda^4_{\mathtt{i}} \supseteq \Lambda^3_{\mathtt{i}} \bullet (F|0) \\
\Lambda^1_{\mathtt{k}} \supseteq \Lambda^3_{\mathtt{k}} \bullet (G|1) & \Lambda^4_{\mathtt{k}} \supseteq \Lambda^3_{\mathtt{k}} \bullet (F|0) \\
\Lambda^2_{\mathtt{i}} \supseteq \Lambda^1_{\mathtt{i}} \bullet (B|0) & \Lambda^1_{\mathtt{z}} \supseteq \Lambda^0_{\mathtt{z}} \bullet (I|0) \\
\Lambda^2_{\mathtt{k}} \supseteq \Lambda^1_{\mathtt{k}} \bullet (B|0) & \Lambda^1_{\mathtt{z}} \supseteq \Lambda^3_{\mathtt{z}} \bullet (G|0) \\
\Lambda^3_{\mathtt{i}} \supseteq \Lambda^2_{\mathtt{k}} \bullet (D|0) & \Lambda^2_{\mathtt{z}} \supseteq \Lambda^1_{\mathtt{z}} \bullet (B|0) \\
\Lambda^3_{\mathtt{i}} \supseteq \Lambda^3_{\mathtt{i}} \bullet (d|2) & \Lambda^3_{\mathtt{z}} \supseteq \Lambda^2_{\mathtt{z}} \bullet (D|0) \\
\Lambda^3_{\mathtt{k}} \supseteq \Lambda^2_{\mathtt{k}} \bullet (D|0) & \Lambda^3_{\mathtt{z}} \supseteq \Lambda^2_{\mathtt{z}} \bullet (d|0) \\
\Lambda^3_{\mathtt{k}} \supseteq \Lambda^2_{\mathtt{k}} \bullet (d|0) & \Lambda^4_{\mathtt{z}} \supseteq \Lambda^3_{\mathtt{z}} \bullet (F|0)
\end{array}$$

Gathering all equations generated from (2.8), (2.10) and (2.12) yields a system $(\mathcal{S})$ of $n_v \times n_s$ equations with $n_v \times n_s$ unknowns, where n_v is the number of induction variables, including `z`, and n_s the number of statements in the program.[6]

`Toy` yields the system

[6] Some unknown sets correspond to variables that are not bound at the node of interest, they are useless.

$$\begin{aligned}
\Lambda^0_{\mathtt{i}} &= \mathcal{E}\\
\Lambda^0_{\mathtt{k}} &= \mathcal{E}\\
\Lambda^0_{\mathtt{z}} &= \mathcal{E}\\
\Lambda^1_{\mathtt{i}} &= \Lambda^3_{\mathtt{i}} \bullet (G|0) + (I|0)\\
\Lambda^1_{\mathtt{k}} &= \Lambda^3_{\mathtt{k}} \bullet (G|1) + (I|0)\\
\Lambda^2_{\mathtt{i}} &= \Lambda^1_{\mathtt{i}} \bullet (B|0)\\
\Lambda^2_{\mathtt{k}} &= \Lambda^1_{\mathtt{k}} \bullet (B|0)
\end{aligned}$$

$$\begin{aligned}
\Lambda^3_{\mathtt{i}} &= \Lambda^3_{\mathtt{i}} \bullet (d|2) + \Lambda^2_{\mathtt{k}} \bullet (D|0)\\
\Lambda^3_{\mathtt{k}} &= \Lambda^3_{\mathtt{k}} \bullet (d|0) + \Lambda^2_{\mathtt{k}} \bullet (D|0)\\
\Lambda^4_{\mathtt{i}} &= \Lambda^3_{\mathtt{i}} \bullet (F|0)\\
\Lambda^4_{\mathtt{k}} &= \Lambda^3_{\mathtt{k}} \bullet (F|0)\\
\Lambda^1_{\mathtt{z}} &= \Lambda^3_{\mathtt{z}} \bullet (G|0) + (I|0)\\
\Lambda^2_{\mathtt{z}} &= \Lambda^1_{\mathtt{z}} \bullet (B|0)\\
\Lambda^3_{\mathtt{z}} &= \Lambda^2_{\mathtt{z}} \bullet (D|0) + \Lambda^2_{\mathtt{z}} \bullet (d|0)\\
\Lambda^4_{\mathtt{z}} &= \Lambda^3_{\mathtt{z}} \bullet (F|0)
\end{aligned}$$

Let Λ be the set of unknowns for $(\mathcal{S})$, i.e., the set of $\Lambda^n_{\mathtt{x}}$ for all induction variables $\mathtt{x}$ and nodes n in the control automaton. Let C be the set of constant coefficients in the system. $(\mathcal{S})$ is a *left linear system of equations over* (Λ, C) [RS97a]. Let X_i be the unknown in Λ appearing in the left-hand side of the i^{th} equation of $(\mathcal{S})$. If $+$ denotes the union in B_{rat}, we may rewrite the system in the form

$$\forall i \in \{1,\ldots,m\}, X_i = \sum_{j=1}^{m} X_j \bullet C_{i,j} + R_i, \tag{2.16}$$

where R_i results from the terms $\Lambda^0_{\mathtt{x}} = \mathcal{E}$ in right-hand side. Note that $C_{i,j}$ is either $\emptyset$ or a bilabel singleton of B_{rat}. Thus $(\mathcal{S})$ is a *strict* system, and as such, it has a unique solution [RS97a]; moreover, this solution can be characterized by a *rational expression* for each unknown set in Λ.

Definition 15 (Rational Function) *If M and M' are two monoids, a* rational function *is a function from M to M' whose graph is a rational relation.*

We may conclude that the solution of $(\mathcal{S})$ is a characterization of each unknown set X_i in Λ as a rational function.

Lemma 4 *For any induction variable $\mathtt{x}$ and node n in the control automaton, the binding function for $\mathtt{x}$ restricted to $\mathcal{L}_n$ $\Lambda^n_{\mathtt{x}}$ is a rational function.*

Theorem 7 *For any induction variable $\mathtt{x}$, the binding function for $\mathtt{x}$ $\Lambda_{\mathtt{x}}$ is a rational function.*

The Theorem is a corollary of Lemma 4, since the functions $\Lambda^n_{\mathtt{x}}$ are defined on disjoint subsets of control words, partitioned according to the suffix n.

Properties of rational relations and functions are similar to those of rational languages [Ber79]: membership, inclusion, equality, emptiness and finiteness are decidable, projection on the input or output monoid yields a rational sub-monoid, and rational relations are closed for union, star, product and inverse morphism, to cite only the most common properties. The main difference is that they are not closed for complementation and intersection, although a useful sub-class of rational relations has this closure property — independently discovered in [PS99] and [Coh99]. Since most of these properties are associated with polynomial algorithms, binding functions can be used in many analyses, see [CC98, Fea98, Coh99, ACF03] for our previous and ongoing applications to the automatic parallelization of recursive programs.

2.5 Computing Binding Functions

This section investigates the resolution of $(\mathcal{S})$. Starting from (2.16), one may compute the last unknown in terms of others:

$$X_m = C^*_{m,m}\Big(\sum_{i=1}^{m-1} X_j \bullet C_{i,j} + R_m\Big). \tag{2.17}$$

The solution of $(\mathcal{S})$ can be computed by iterating this process analogous to Gaussian elimination. This was the first proposed algorithm [Coh99]; but Gaussian elimination on non-commutative semirings leads to exponential space requirements. We propose two alternative methods to compute and represent binding functions effectively. The first one improves on Gaussian elimination but keeps an exponential complexity; its theoretical interest is to capture the *relations between all induction variables* along a single path on the control automaton. If we only need to represent the computation of induction variables *separately* from each other, Section 2.5.2 presents a polynomial algorithm.

2.5.1 Binding Matrix

M_{rat} denotes the set $B_{\text{rat}}^{m\times m}$ of square matrices of dimension m with elements in B_{rat}; M_{rat} is a semiring for the induced matrix addition and product and M_{rat} is closed by star operation [RS97a]. The neutral element of M_{rat} is

$$\mathbb{E} = \left[\begin{array}{ccc} \mathcal{E} & & \emptyset \\ & \ddots & \\ \emptyset & & \mathcal{E} \end{array}\right]. \tag{2.18}$$

Practical computation of the transitive closure of a square matrix C is an inductive process, using the following block decomposition where a and d are square matrices:

$$C = \left[\begin{array}{cc} a & c \\ b & d \end{array}\right].$$

The formula is illustrated by the finite-state automaton in Figure 2.10; its alphabet is the set of labels $\{a,b,c,d\}$ of the block matrices; i and j are the two states, they are both initial and final. If i and j denote the languages computed iteratively for the two states, and matrix C represents a linear transformation of the vector (i,j): $(i_1,j_1) = (i_0a+j_0b, i_0c+j_0d)$. We compute the transitive closure of C as the union of all words labeling a path terminated in states i or j, respectively, after zero, one, or more applications of C: $(i_*,j_*) = \big((i_0+j_0d^*b)(a+cd^*b)^*, (j_0+i_0a^*c)(d+ba^*c)^*\big)$. Writing $P=(a+cd^*b)^*$ and $Q=(d+ba^*c)^*$,

$$C^* = \left[\begin{array}{cc} a & c \\ b & d \end{array}\right]^* = \left[\begin{array}{cc} P & d^*bP \\ a^*cQ & Q \end{array}\right]. \tag{2.19}$$

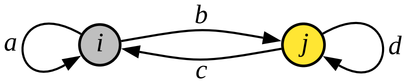


Figure 2.10: Computation of a matrix star

From (2.16), system $(\mathcal{S})$ can be written $X = XC+R$, where (matrix) $C = (C_{i,j})_{1\le i,j\le m}$ and (vectors) $R = (R_1,\ldots,R_m)$, $X = (X_1,\ldots,X_m)$. Vector RC^* is the solution of $(\mathcal{S})$, but direct application of (2.19) is still laborious, given the size of C.

Matrix Automaton

Our solution relies on the sparsity of C: we represent the system of equations in the form of an automaton $\mathcal{A}$, called the matrix automaton.

The graph of the matrix automaton is the same as the graph of the control automaton. Each statement s is represented by a unique transition, gathering all information about induction variable updates while executing s. The *binding function for* `x` *after statement* s, $\Lambda_{s\texttt{x}}$, maps control words ended by s to the value of `x`. It is the set of all possible bindings for `x` after s. $\overrightarrow{\Lambda^n}$ denotes the *binding vector at state* n, i.e., the tuple of binding functions for all induction variables at state n (including `z`). Conversely, $\overrightarrow{\Lambda_s}$ denotes the *binding vector after statement* s, i.e., the tuple of binding functions for all induction variables after executing statement s.

With d the departure state of the transition associated to statement s, we gather the previous linear equations referring to s and present them in the form:

$$\forall \mathbb{S} \in M_{\text{rat}}, \overrightarrow{\Lambda_s} = \overrightarrow{\Lambda^d} \times \mathbb{S}. \tag{2.20}$$

As an example, we give the result for statement G of `Toy`:

$$\Lambda_{G\texttt{i}} = \Lambda^3_{\texttt{i}} \bullet (G|0),\ \Lambda_{G\texttt{k}} = \Lambda^3_{\texttt{k}} \bullet (G|1),\ \Lambda_{G\texttt{z}} = \Lambda^3_{\texttt{z}} \bullet (G|0)$$

$$\overrightarrow{\Lambda_G} = \overrightarrow{\Lambda^3} \times \left[\begin{array}{ccc} (G|0) & \emptyset & \emptyset \\ \emptyset & (G|1) & \emptyset \\ \emptyset & \emptyset & (G|0) \end{array}\right].$$

Now, the transition of statement s in $\mathcal{A}$ is labeled by the *statement matrix* $\mathbb{S}$. Thus, $\mathcal{A}$ recognizes words with alphabet in M_{rat}: concatenation is the matrix product and words are rational expression in M_{rat}, hence elements of M_{rat}. Grouping equations according to the transitions' arrival state, we get, for each state a,

$$\overrightarrow{\Lambda^a} = \sum_{d \in \text{pred}(a)} \overrightarrow{\Lambda^d} \times \mathbb{S}_{da}, \mathbb{S}_{da} \in M_{\text{rat}}, \tag{2.21}$$

where $\text{pred}(a)$ is the set of predecessor states of a and $\mathbb{S}_{da}$ is the statement matrix associated to the transition from d to a.

E.g., state number 1 in the matrix automaton of `Toy` yields

$$\overrightarrow{\Lambda^1} = \overrightarrow{\Lambda_I} + \overrightarrow{\Lambda_G} = \overrightarrow{\Lambda^0} \times \mathbb{I} + \overrightarrow{\Lambda^3} \times \mathbb{G}.$$

Theorem 8 *Let $\overrightarrow{\Lambda^0} = (\mathcal{E}, \ldots, \mathcal{E})$ be the binding vector at the beginning of the execution. The binding vector for any state f can be computed as*

$$\overrightarrow{\Lambda^f} = \overrightarrow{\Lambda^0} \times \mathbb{L}, \tag{2.22}$$

where $\mathbb{L}$ is a matrix of regular expressions of bilabels; $\mathbb{L}$ is computed from the regular expression associated to the matrix automaton $\mathcal{A}$, when its unique final state is f.

This result is a corollary of Theorem 7.

Because this method operates on regular expressions, it has a worst-case exponential complexity in the number of states and induction variables. However, this worst-case behavior is not likely on typical examples.

Application to the Running Example

We now give the statement matrices associated with equations (2.11) to (2.15). With the three induction variables `i`, `k` and `z`, the binding vector after statement I, $\overrightarrow{\Lambda_I} = (\Lambda_{Ii}, \Lambda_{Ik}, \Lambda_{Iz})$ and $\mathbb{I}$ the statement matrix for I, we have:

$$\begin{array}{lll} \overrightarrow{\Lambda_I} = \overrightarrow{\Lambda^0} \times \mathbb{I}, & \overrightarrow{\Lambda_B} = \overrightarrow{\Lambda^1} \times \mathbb{B}, & \overrightarrow{\Lambda_D} = \overrightarrow{\Lambda^2} \times \mathbb{D} \\ \overrightarrow{\Lambda_d} = \overrightarrow{\Lambda^3} \times {\scriptstyle\mathbb{D}}, & \overrightarrow{\Lambda_G} = \overrightarrow{\Lambda^3} \times \mathbb{G}, & \overrightarrow{\Lambda_F} = \overrightarrow{\Lambda^3} \times \mathbb{F} \end{array}$$

with the following statement matrices:

$$\text{statement } I: \quad \mathbb{I} = \begin{bmatrix} I|0 & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \\ \emptyset & I|0 & I|0 \end{bmatrix}$$

$$\text{statement } G: \quad \mathbb{G} = \begin{bmatrix} G|0 & \emptyset & \emptyset \\ \emptyset & G|1 & \emptyset \\ \emptyset & \emptyset & G|0 \end{bmatrix}$$

$$\text{statement } d: \quad {\scriptstyle\mathbb{D}} = \begin{bmatrix} d|2 & \emptyset & \emptyset \\ \emptyset & d|0 & \emptyset \\ \emptyset & \emptyset & d|0 \end{bmatrix}$$

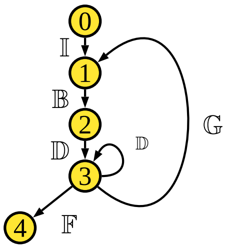
$$\text{statement } D: \quad \mathbb{D} = \begin{bmatrix} \emptyset & \emptyset & \emptyset \\ D|0 & D|0 & \emptyset \\ \emptyset & \emptyset & D|0 \end{bmatrix}$$

The other statements matrices let unchanged the induction variables.

$$\text{statement } B: \quad \mathbb{B} = \begin{bmatrix} B|0 & \emptyset & \emptyset \\ \emptyset & B|0 & \emptyset \\ \emptyset & \emptyset & B|0 \end{bmatrix}$$

$$\text{statement } F: \quad \mathbb{F} = \begin{bmatrix} F|0 & \emptyset & \emptyset \\ \emptyset & F|0 & \emptyset \\ \emptyset & \emptyset & F|0 \end{bmatrix}$$

The resulting matrix automaton is shown in Figure 2.11 (all states are final).



$$\mathbb{L} = \mathbb{I} + \mathbb{IB} + \mathbb{IBD}\left(\mathbb{D}+\mathbb{GBD}\right)^{*}\left(\mathbb{E}+\mathbb{F}+\mathbb{G}+\mathbb{GB}\right)$$

($\mathbb{E}$ is the neutral element of M_{rat}.)

Figure 2.11: Example of matrix automaton

2.5.2 Binding Transducer

We recall a few definitions and results about transducers [Ber79].

A *rational transducer* is a finite-state automaton where each transition is labeled by a pair of *input* and *output* symbols (borrowing from Definition 12), a symbol being a letter of the alphabet or the empty word.[7]

A pair of words (u,v) is *recognized* by a rational transducer if there is a path from an initial to a final state whose input word is equal to u and output word is equal to v.[8]

A rational transducer recognizes a rational relation, and reciprocally.

A transducer offers either a static point of view — as a machine that recognizes pairs of words — or a dynamic point of view — the machine reads an input word and outputs the set of image words.

The use of transducers lightens the burden of solving a system of regular expressions, but we lose the ability to capture all induction variables and their relations in a single object. The representation for the binding function of an induction variable is called the *binding transducer*.

Algorithm 2

Given the control automaton and a monoid with n_v induction variables (including `z`*), the binding transducer is built as follows:*

- *For each control automaton state, create a set of n_v states, called a* product-state*; each state of a product-state is dedicated to a specific induction variable.*

- *Initial (resp. final) states correspond to the product-states of all initial (resp. final) states of the control automaton.*

- *For each statement s, i.e., for each transition (d,a) labeled s in the control automaton; call P^d and P^a the corresponding product-states; and create an associated* product-transition *t_s. It is a set of n_v transitions, each one is dedicated to a specific induction variable. We consider again the two cases mentioned in Definition (10.**c**).*

  - *case **c**.1: the transition runs from state $P^d_{\mathtt{z}}$ in P^d to the state $P^a_{\mathtt{x}}$ in P^a. The input label is s, the output label is the initialization constant c;*
  - *case **c**.2: the transition runs from state $P^d_{\mathtt{y}}$ in P^d to state $P^a_{\mathtt{x}}$ in P^a. The input label is s, the output label is the constant c.*

The binding transducer for `Toy` is shown in Figure 2.12. Notice that nodes allocated to the virtual induction variable `z` are not co-accessible except the initial state (there is no path from them to a final state), and initial states dedicated to `i` and `k` are not co-accessible either. These states are useless, they are trimmed from the binding transducer.

The binding transducer does not directly describe the binding functions. A binding transducer is *dedicated* to an induction variable `x` when its final states are restricted to the states dedicated to `x` in the final product-states.

Theorem 9 *The binding transducer dedicated to an induction variable* `x` *recognizes the binding function for* `x`.

This result is a corollary of Theorem 7.

[7] Pair of words leads to an equivalent definition.

[8] A transducer is not reducible to an automaton with bilabels as elementary symbols for its alphabet; as an illustration, two paths labeled $(x|\varepsilon)(y|z)$ and $(x|z)(y|\varepsilon)$ recognize the same pair of words $(xy|z)$.

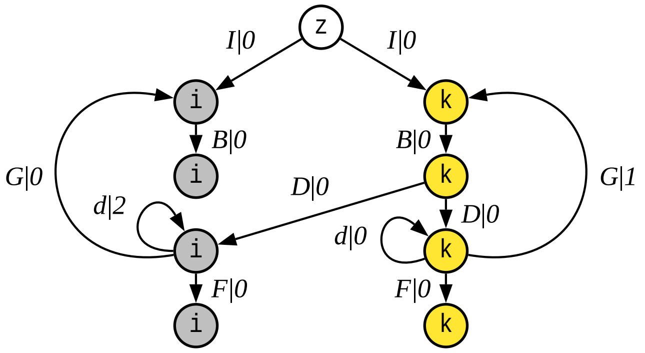


Figure 2.12: Binding Transducer for `Toy`

2.6 Experiments

The construction of the binding transducer is fully implemented in OCaml. Starting from a MOGUL program, the analyzer returns the binding transducer according to the choice of monoid. This analyzer is a part of a more ambitious framework including dependence test algorithms based on the binding transducer [ACF03]. Our implementation is as generic as the framework for data structure and binding function computation: operations on automata and transducers are parameterized by the types of state names and transition labels. Graphs of automata and transducers are drawn by the free `dot` software [KN02].

We present two examples processed by our instancewise analyzer of MOGUL programs. The first one operates on an array, the second one on a tree.

The `Pascaline` Program

Figure 2.13 shows a program to evaluate the binomial coefficients (a line of Pascal's triangle). It exhibits both a loop statement and a recursive call, two induction variables `I` and `L` plus the constant induction variable `n`; `x` and `y` are not induction variables. Statement *D*, `x = 1`, is an elementary statement without induction variables: MOGUL simply ignores it. The `else` branch of the conditional is empty: it ensures the termination of recursive calls.

```
  structure Monoid_int A;

A function Pascaline(Monoid_int L, Monoid_int n) {
    int x, y;
B   if (L < n)
C   {
D     x = 1;
E     for (Monoid_int I=1; I<n;
e          I=I.1)
F     {
G         y = A[I];
H         A[I] = x + y;
I         x = A[I];
      }
J     Pascaline(L.1, n);
    }
  }

K function Main() {
L   Pascaline(0, 10);
  }
```

Figure 2.13: Program `Pascaline`

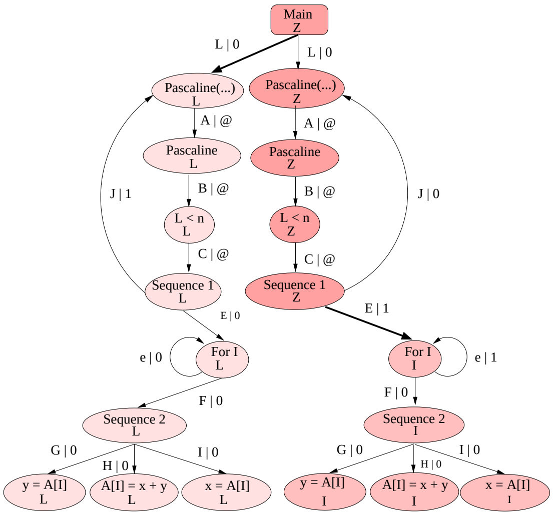


Figure 2.14: Binding transducer for `Pascaline`

Figure 2.14 shows the binding transducer for `Pascaline`, as generated by the software. The transducer is drawn by hand to enhance readability, and in complement with the indication of the dedicated induction variable, we filled each node of the graph with a statement borrowed from the program: the statement is written in the arrival nodes of the associated transitions. Nodes dedicated to the induction variable `n` are not used; they have been trimmed. Notice the use of induction variable `z` to initialize loop counter `I`.

```
  monoid Monoid_tree [next, left, right];
  structure Monoid_tree Tree;

t function Main() {
s   Sort(@, 37);
  }

S function Split(Monoid_tree A,
                 Monoid_tree B,
                 Monoid_tree C,
                 Monoid_int n) {
F   if (n>0)
B   {
A     Tree[B] = Tree[A];
    }
L   if (n>1)
H   {
G     Tree[C] = Tree[A.next];
    }
R   if (n>2)
N   {
M     Split(A.next.next, B.next, C.next,
            n-2);
    }
  }
```

```
h function Merge(Monoid_tree A,
                 Monoid_tree B,
                 Monoid_tree C,
                 int p, int q) {
g   if ((q != 0)
         && (p = 0 || Tree[B] < Tree[C]))
V   {
T     Tree[A] = Tree[B];
U     Merge(A.next, B.next, C, q-1, p );
    }
e   else
d   {
c     if (p != 0)
Y     {
W       Tree[A] = Tree[C];
X       Merge(A.next, B, C.next, q, p-1);
      }
    }
  }

r function Sort(Monoid_tree T, int r) {
q   if (r > 1)
m   {
i     Split(T, T.left, T.right, r);
j     Sort(T.left, (r+1)/2);
k     Sort(T.right, r/2);
l     Merge(T, T.left, T.right,(r+1)/2,
            r/2);
    }
  }
```

Figure 2.15: Program `Merge_sort_tree`

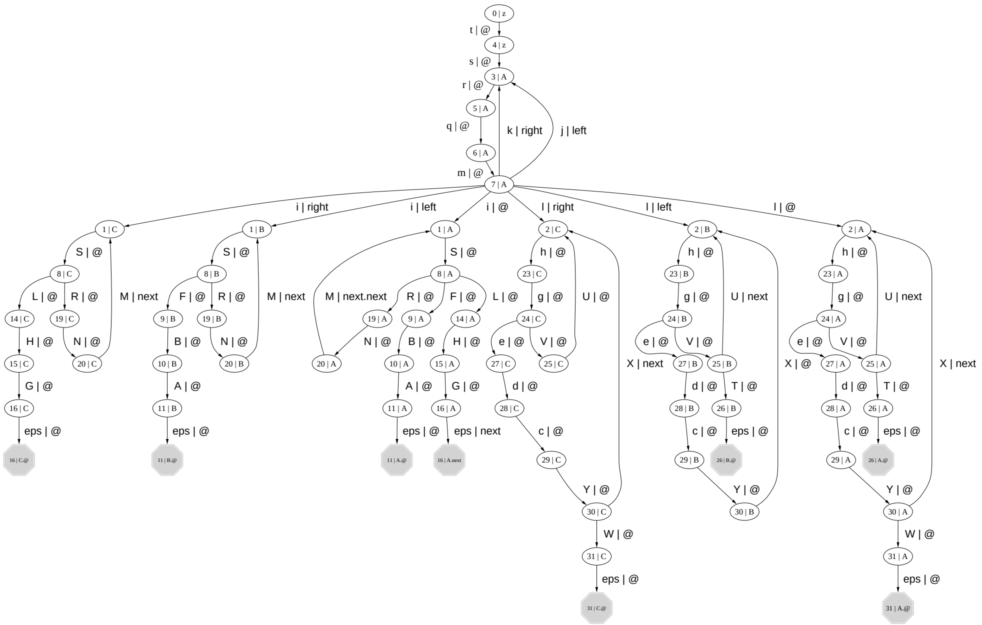


Figure 2.16: Binding transducer for `Merge_sort_tree`

The `Merge_sort_tree` Program

Figure 2.15 shows an implementation of the merge sort algorithm, implemented over a binary tree of lists, called `Tree`. The three functions `Split`, `Merge` and `Sort` are recursive. Induction variables `A`, `B` and `C` are locations in the tree; they are overloaded and exchanged as formal parameters of the three functions. Parameter `n` of `Split` is an independent induction variable not used for memory accesses, and `p`, `q` and `r` are not induction variables. `@` denotes the empty word, i.e., the root of the tree.

The binary tree of lists is represented as a ternary tree: `next` is the field for the first branch, it traverses a list of integers, the `left` and `right` fields traverse the backbone binary tree. At the beginning, the unsorted list is stored in the `next` branch of the tree named *Tree*. It is split in two halves stored in the `left` and `right` branches. Both these lists are recursively sorted, then merged back in the root node. Figure 2.16 shows the binding transducer for `Merge_sort_tree` as drawn by `dot` [KN02] from the MoGuL software output. Octogonal states correspond to the tree references at the elementary statements. These states are useful for the computation of data dependences. Indeed, from this binding transducer, we developed algorithms to detect that the two calls to the `Sort` function (*j* and *k*) can be run in parallel [Fea98, Coh99].

Other sample programs

Figure 2.17 summarizes some results about recursive programs we implemented in MoGuL. The last column of the table gives the number of states in the binding transducer. Such a binding transducer can been used to check for dependences, uninitialized values, opportunities for instancewise dead-code elimination and other instancewise extensions of bit-vector analyses, etc.

Since the first survey of instancewise analyses techniques [Coh99], we discovered many recursive algorithms suitable for implementation in MoGuL and instancewise dependence analysis. Therefore, it seems that the program model encompasses many implementations of practical algorithms despite its severe constraints.

Program `n-Queens` is the classical problem to place *n* Queens on a $n \times n$ chessboard. `To_&_fro` is the recursive merge-sort algorithm alternating over two arrays. It is optimized in `To_&_fro+Terminal_insert_sort` by using an insertion sort for the leaves of the recursion (on small intervals of the original array). `Sort_3_colors` consists in sorting an array of balls according to one color among three, using only swaps. `Vlsi_test` simulates a test-bed to filter-out good chips from an array of untested ones; the process relies on peer-to-peer test of two chips, a good chip giving a certified correct answer about the other.

Code name	Data	Lines	Data Refs	Loops	Fn Calls	Nodes
`Pascaline`	1D array	21	2	1	2	13
`Multiplication table`	2D array	17	5	1	3	22
`n-Queens`	1D array	39	2	2	2	27
`To_&_fro`	1D array	115	12	0	19	164
`Merge_sort_tree`	ternary tree	75	8	0	8	80
`To_&_fro+Terminal_insert_sort`	1D array	162	17	2	26	195
`Sort_3_colors`	1D array	80	4	0	11	97
`Vlsi_test`	linked lists	58	2	0	7	97

Figure 2.17: Sample recursive programs applicable to binding function analysis

2.7 Applications of Instancewise Analysis

To illustrate the practical applications of the binding transducer, we describe a very simple program optimization that benefit from the computation of instancewise binding functions: *instancewise dead-code elimination*, then we outline the main results in the area.

2.7.1 Instancewise Dead-Code Elimination

Going back to the motivating examples in Section 1.2 (either code version), we call *s* the assignment to array `A` in the loop nest and *t* the read reference in procedure `line`. We assume the codes have been rewritten in MoGuL (the first version has a single induction variable addressing $\mathbb{Z}^2$ and the second one has two recursively swapped induction variables).

Let B_s and B_t denote the binding functions for the array references in s and t, respectively, and L^*_{ab} denote the set of all control words. The “chessboard” footprint, very hard to compute by statementwise means, corresponds to the rational set $B_t(L^*_{\text{ab}})$. An intensional representation for this rational set can be computed, either as a finite-state automaton (in a straightforward transducer projection [Ber79]), or as a $\mathbb{Z}$-polyhedron (e.g., through a Parikh mapping [Par66, RS97a]).

From this first result, one may automatically characterize the iterations of the loop nest which correspond to useless assignments to A: the (conservative) set of dead iterations is $B_s^{-1}(B_t(L^*_{\text{ab}}))$. Once again, this turns out to be a classical operation on transducers and finite-state automata. To implement the actual optimization on the bounds and strides, a polyhedral characterization of the iteration domain can be deduced from the resulting automaton (because s is surrounded by a loop nest, not arbitrary recursive control) [PD96, RS97a].

2.7.2 State of the Art

Dead-code elimination is a very simple application of the instancewise framework. One may imagine many other extensions of classical scalar, loop and interprocedural optimizations, working natively on recursive programs. However, published results are still preliminary and rather off the tracks of most work on static analysis [CC98, Fea98, Coh99, ACF03]: here is a short overview of the known applications of binding functions to the analysis of recursive programs.

- Instancewise dependence analysis for arrays [CC98, Coh99]. The relation between dependent instances is computed as a one-counter (context-free) transducer, or by a multi-counter transducer in the case of multi-dimensional arrays. In the multi-counter case, the characterization of dependences is undecidable in general, but approximations are possible.

- Instancewise reaching-definition analysisfor arrays [CC98, Coh99] (a.k.a. array data-flow analysis [Fea91, MAL93]). Compared to dependence analysis, kills of previous array assignments are taken into account. Due to the conservative assumptions about conditional guards, one may only exploit kill information based on structural properties of the program, i.e., exclusive branches and ancestry of control words in the call tree (whether an instance forcibly precedes another in the execution). This limitation seems rather strong, but it already subsumes the loop-nest case [Coh99].

- Instancewise dependence and reaching-definition analysis for trees [Coh99]. The relation between conflicting instances is a rational transducer, from the Elgot and Mezei theorem [EM65, Ber79]; the dependence relation requires an additional sequentiality constraint, which makes its characterization undecidable in general, but an approximation scheme based on synchronous transducers is available [PS99, Coh99]. The array and tree cases can be unified: [Coh99] describes a technique to analyze nested trees and arrays in free partially-commutative monoids [RS97b].

- Instancewise dependence test for trees [Fea98]. Instead of a relation between instances, this test leverages on instancewise analysis to compute precise statementwise dependence information with unlimited context-sensitivity (not k-limited). This technique features a semi-algorithm to solve the undecidable dependence problem, and the semi-algorithm is proven to terminate provided the approximation scheme of the previous technique is used (unpublished result).

- Instancewise dependence test for arrays [ACF03, Ami04]. Amiranoff’s thesis proves the decidability and NP-completeness of dependence testing based on binding transducers, in the case of arrays. An extension taking conditional guards into account is available, provided the guards can be expressed as affine functions of some inductive variables lying in free-commutative monoids (unpublished result). This extension defines conditions for the exactness of the dependence test (i.e., the absence of approximation) that strictly generalize the case of static-control loop nests.

2.8 Conclusion

The instancewise paradigm paves the way for better, more precise program analyses. It decouples static analyses from the program syntax, allowing to evaluate semantic program properties on an infinite set of runtime control points. This paradigm abstracts runtime execution states (or trace prefixes) in a finitely-presented, infinite set of control words. Instancewise analysis is also an extension of the domain-specific iteration-vector approach (the so-called polytope model) to general recursive programs.

As an application of the instancewise framework, we extend the concept of induction variables to recursive programs. For a restricted class of data structures (including arrays and recursive structures), induction variables capture the exact memory location accessed at every step of the execution. This compile-time characterization, called the binding function, is a rational function mapping control words to abstract memory locations. We give a polynomial algorithm for the computation of binding functions.

Our current work focuses on instancewise alias and dependence analysis, for the automatic parallelization and optimization of recursive programs [Ami04]. We also look after new benchmark applications and data-structures to assess the applicability of binding functions; multi-grid and sparse codes are interesting candidates. We would also like to release a few constraints on the data structures and induction variables, aiming for the computation of approximate binding functions through abstract interpretation.

Chapter 3

Polyhedral Program Manipulation

This chapter presents the technical background and contributions of our polyhedral, semantics-based program representation. The use of polyhedral domains to capture both the control and data flow allows to abstract away many implementation artifacts of syntax-based representations, and to define most loop transformations without reference to any syntactic form of the program.

Structure of the chapter. Section 3.1 illustrates with a simple example the limitations of syntactic representations for transformation composition, it presents our polyhedral representation and how it can circumvent these limitations. Revisiting classical loop transformations for automatic parallelization and locality enhancement, Section 3.2 generalizes their definitions in our framework, extending their applicability scope, abstracting away most syntactic limitations to transformation composition, and facilitating the search for compositions of transformations. Using several SPEC benchmarks, Section 3.3 shows that complex compositions can be necessary to reach high performance and how such compositions are easily implemented using our polyhedral representation. Section 3.4 describes the implementation of our representation, of the associated transformation tool, and of the code generation technique (in Open64/ORC [ORC]). Section 3.5 validates these tools through the evaluation of a dedicated transformation sequence for one benchmark. Section 3.6 presents more algorithmic research on reducing the complexity of finding a sequence of loop transformations. Every section discusses the closest technical work, but Section 3.7 summarizes work that relate to the overall approach and infrastructure.

3.1 A New Polyhedral Program Representation

The purpose of Section 3.1.1 is to illustrate the limitations of the implementation of program transformations in current compilers, using a simple example. Section 3.1.2 is a gentle introduction to polyhedral representations and transformations. In Section 3.1.3, we present our polyhedral representation, in Section 3.1.4 how it alleviates syntactic limitations and Section 3.1.5 presents normalization rules for the representation.

3.1.1 Limitations of Syntactic Transformations

In current compilers, after applying a program transformation to a code section, a new version of the code section is generated, using abstract syntax trees, three address code, SSA graphs, etc. We use the term *syntactic* (or syntax-based) to refer to such transformation models. Note that this behavior is also shared by all previous matrix- or polyhedra-based frameworks.

Code size and complexity

As a result, after multiple transformations the code size and complexity can dramatically increase.

Consider the simple synthetic example of Figure 3.1, where it is profitable to merge loops i,k (the new loop is named i), and then loops j,l (the new loop is named j), to reduce the locality distance of array `A`, and then to tile loops i and j to exploit the spatial and TLB locality of array `B`, which is accessed column-wise. In order to perform all these transformations, the following actions are necessary: merge loops i, k, then merge loops j, l, then split statement `Z[i]=0` outside the i loop to enable tiling, then strip-mine loop j, then strip-mine loop i and then interchange i and jj (the loop generated from the strip-mining of j).

```
     for (i=0; i<M; i++)
S1     Z[i] = 0;
       for (j=0; j<N; j++)
S2       Z[i] += (A[i][j] + B[j][i]) * X[j];
     for (k=0; k<P; k++)
       for (l=0; l<Q; l++)
S3       Z[k] += A[k][l] * Y[l];
```

Figure 3.1: Introductory example

	Syntactic	(#lines)	Polyhedral	(#values)
Original code	11		78	
Outer loop fusion	44	(×4.0)	78	(×1.0)
Inner loop fusion	132	(×12.0)	78	(×1.0)
Fission	123	(×11.2)	78	(×1.0)
Strip-Mine	350	(×31.8)	122	(×1.5)
Strip-Mine	407	(×37.0)	182	(×2.3)
Interchange	455	(×41.4)	182	(×2.3)

Figure 3.2: Code size versus representation size

	Original	KAP	Double Fusion	Full Sequence
Time (s)	26.00	12.68	19.00	7.38

Figure 3.3: Execution time

```
...;
if ((M >= P+1) && (N == Q) && (P >= 63))
  for (ii=0; ii<P-63; ii+=64)
    for (jj=0; jj<Q; jj+=64)
      for (i=ii; i<ii+63; i++)
        for (j=jj; j<min(Q,jj+63); j++)
          Z[i] += (A[i][j] + B[j][i]) * X[j];
          Z[i] += A[i][j] * Y[j];
  for (ii=P-62; ii<P; ii+=64)
    for (jj=0; jj<Q; jj+=64)
      for (i=ii; i<P; i++)
        for (j=jj; j<min(Q,jj+63); j++)
          Z[i] += (A[i][j] + B[j][i]) * X[j];
          Z[i] += A[i][j] * Y[j];
      for (i=P+1; i<min(ii+63,M); i++)
        for (j=jj; j<min(N,jj+63); j++)
          Z[i] += (A[i][j] + B[j][i]) * X[j];
  for (ii=P+1; ii<M; ii+=64)
    for (jj=0; jj<N; jj+=64)
      for (i=ii; i<min(ii+63,M); i++)
        for (j=jj; j<min(N,jj+63); j++)
          Z[i] += (A[i][j] + B[j][i]) * X[j];
...;
```

Figure 3.4: Versioning after outer loop fusion

Because the *i* and *j* loops have different bounds, the merging and strip-mining steps will progressively multiply the number of loop nests versions, each with a different guard. After all these transformations, the program contains multiple instances of the code section shown in Figure 3.4. The number of program statements after each step is indicated in Figure 3.2.

In our framework, the final generated code will be similarly complicated, but this complexity does not show until code generation, and thus, it does not hamper program transformations. The polyhedral program representation consists in a fixed number of matrices associated with each statement, and neither its complexity nor its size vary significantly, independently of the number and nature of program transformations. The number of statements remains the same (until the code is generated), only some matrix dimensions may increase slightly, see Figure 3.2. Note that the more complex the code, the higher the difference: for instance, if the second loop is triangular, i.e., `(j=0; j<i; j++)`, the final number of source lines of the syntactic version is 34153, while the size of the polyhedral representation is unchanged (same number of statements and matrix dimensions).

Breaking patterns

Compilers look for transformation opportunities using pattern-matching rules. This approach is fairly fragile, especially in the context of complex compositions, because previous transformations may break target patterns for further ones. Interestingly, this weakness is confirmed by the historical evolution of the SPEC CPU benchmarks themselves, partly driven by the need to avoid pattern-matching attacks from commercial compilers [PJEJ04].

To illustrate this point we have attempted to perform the above program transformations targeting the Alpha 21364 EV7, using KAP C (V4.1) [KAP], one of the best production preprocessors available (source to source loop and array transformations). Figure 3.3 shows the performance achieved by KAP and by the main steps of the above sequence of transformations (fusion of the outer and inner loops, then tiling) on the synthetic example.[1] We found that KAP could perform almost none of the above transformations because pattern-matching rules were often too limited. Even though we did not have access to the KAP source code, we have reverse-engineered these limitations by modifying the example source code until KAP could perform the appropriate transformation; KAP limitations are deduced from the required simplifications. In Section 3.1.4, we will show how these limitations are overridden by the polyhedral representation.

The first step in the transformation sequence is the fusion of external loops i, k: we found that KAP only attempts to merge perfectly nested loops with matching bounds (i.e., apparently due to additional conditions of KAP's fusion pattern); after changing the loop bounds and splitting out `Z[i]=0`, KAP could merge loops i, k. In the polyhedral representation, fusion is only impeded by semantic limitations, such as dependences; non-matching bounds or non-perfectly nested loops are not an issue, or more exactly, these artificial issues simply disappear, see Section 3.1.4. After enabling the fusion of external loops i, k, the second step is the fusion of internal loops j, l. Merging loops j, l changes the ordering of assignments to `Z[i]`. KAP refuses to perform this modification (apparently another condition of KAP's fusion pattern); after renaming `Z` in the second loop and later accumulating on both arrays, KAP could perform the second fusion.

Overall, we found out that KAP was unable to perform these two transformations, mostly because of pattern-matching limitations that do not exist in the polyhedral representation. We performed additional experiments on other modern loop restructuring compilers, such as Intel Electron (IA64), Open64/ORC (IA64) and EKOPath (IA32, AMD64, EM64T), and we found similar pattern-matching limitations.

Flexible and complex transformation composition

Compilers come with an ordered set of phases, each phase running some dedicated optimizations and analyses. This phase ordering has a major drawback: it prevents transformations from being applied several times, after some other *enabling* transformation has modified the applicability or adequation of further optimizations. Moreover, optimizers have rather rigid optimization strategies that hamper the exploration of potentially useful transformations.

Consider again the example of Figure 3.1. As explained above, KAP was unable to split statement `Z[i]` by itself, even though the last step in our optimization sequence — tiling j after fusions — cannot be performed without that preliminary action. KAP's documentation [KAP] shows that fission and fusion are performed together (and possibly repeatedly) at a given step in KAP's optimization sequence. So while fission could be a potentially enabling transformation for fusion (though it failed in our case for the reasons outlined in the previous paragraph), it is not identified as an enabling transformation for tiling in KAP's strategy, and it would always fail to split to enable tiling.

[1] Parameters M, N, P and Q are the bounds of the 400MB matrices `A` and `B`.

Moreover, even after splitting `Z[i]` and merging loops *i*, *k* and *j*, *l*, KAP proved unable to tile loop *j*; it is probably focusing on scalar promotion and performs unroll-and-jam instead, yielding a peak performance of 12.35*s*. However, in our transformation sequence, execution time decreases from 26.00*s* to 19.00*s* with fusion and fission, while it further decreases to 7.38*s* thanks to tiling. Notice that both fusion and tiling are important performance-wise.

So KAP suffers from a too rigid optimization strategy, and this example outlines that, in order to reach high performance, a flexible composition of program transformations is a key element. In Section 3.3, we will show that, for one loop nest, up to 23 program transformations are necessary to outperform peak SPEC performance.

Limitations of phase ordering

```
   B[1] = 0
   for (i=0; i<100; i++)
A  |  A[i] = ...;
   for (i=0; i<99; i++)
B  |  B[i+1] = A[i] ...;
   for (i=0; i<100; i++)
C  |  C[i] = B[i] ...;
```

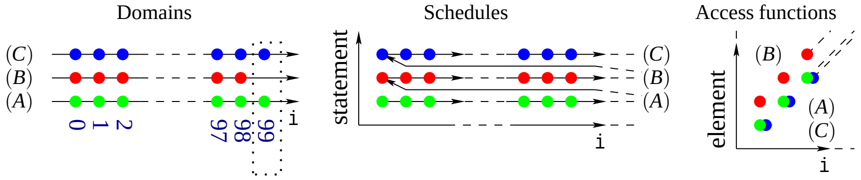


Figure 3.5: Original program and graphical view of its polyhedral representation

```
   B[1] = 0
   for (i=0; i<99; i++)
A  |  a = ...;
B  |  B[i+1] = a ...;
C  |  C[i] = ...;
   C[100] = B[100] ...;
```

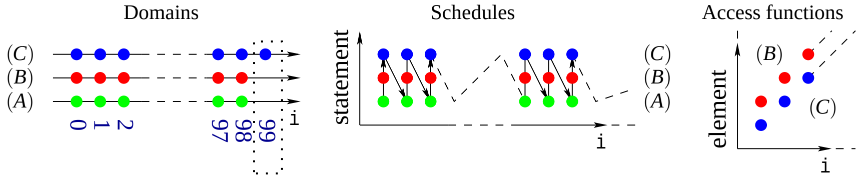


Figure 3.6: Target optimized program and graphical view

To better understand the interplay of loop peeling, loop fusion, scalar promotion and dead-code elimination, let us now consider the simpler example of Figure 3.5. The three loops can be fused to improve temporal locality, and assuming `A` is a local array not used outside the code fragment, it can be replaced with a scalar `a`. Figure 3.6 shows the corresponding optimized code. Both figures also show a graphical representation of the different domains, schedules and access functions for the three statements *A*, *B* and *C* of the original and optimized versions. Notice the middle loop in Figure 3.5 has a reduced domain. These optimizations mainly consist in loop fusions which only have an impact on scheduling, the last iteration (99) in the domain of *A* was removed (dead code) and the access function to array `A` disappeared (scalar promotion).

Again, we tried to optimize this example using KAP, assuming that `A` is a global array, effectively restricting ourselves to peeling and fusion.

```
B[1] = 0
for (i=0; i<99; i++)
 | A[i] = ...;
 | B[i+1] = A[i] ...;
 | C[i] = B[i] ...;
A[100] = ...;
C[100] = B[100] ...;
```

Figure 3.7: Fusion of the three loops

```
B[1] = 0
for (i=0; i<99; i++)
 | A[i] = ...;
 | B[i+1] = A[i] ...;
A[100] = ...;
for (i=0; i<100; i++)
 | C[i] = B[i] ...;
```

Figure 3.8: Peeling prevents fusion

```
B[1] = 0
for (i=0; i<99; i++)
 | a = ...;
 | A[i] = a
 | B[i+1] = a ...;
for (i=0; i<100; i++)
 | C[i] = B[i] ...;
```

Figure 3.9: Dead code before fusion

```
B[1] = 0
for (i=0; i<99; i++)
 | a = ...;
 | B[i+1] = a ...;
for (i=0; i<100; i++)
 | C[i] = B[i] ...;
```

Figure 3.10: Fusion before dead code

```
   B[1] = 0
   for (i=0; i<100; i++)
A   | A[i] = A[1] ...;
   for (i=0; i<99; i++)
B   | B[i+1] = A[i] ...;
   for (i=0; i<100; i++)
C   | C[i] = A[i]+B[i] ...;
```

Figure 3.11: Advanced example

```
B[1] = 0
for (i=0; i<99; i++)
 | A[i] = A[1] ...;
 | B[i+1] = A[i] ...;
 | C[i] = B[i] ...;
A[100] = A[1] ...;
C[100] = A[100]+B[100] ...;
```

Figure 3.12: Fusion of the three loops

```
B[1] = 0
for (i=0; i<99; i++)
 | A[i] = A[1] ...;
 | B[i+1] = A[i] ...;
A[100] = A[1] ...;
for (i=0; i<100; i++)
 | C[i] = A[i]+B[i] ...;
```

Figure 3.13: Spurious dependences

The reduced domain of *B* has no impact on our framework, which succeeds in fusing the three loops and yields the code in Figure 3.7. However, to fuse those loops, syntactic transformation frameworks require some iterations of the first and third loop to be peeled and interleaved between the loops. Traditional compilers are able to peel the last iteration and fuse the first two loops, as shown in Figure 3.8. Now, because their pattern for loop fusion only matches consecutive loops, peeling prevents fusion with the third loop, as shown in Figure 3.8; we checked that neither a failed dependence test nor an erroneous evaluation in the cost model may have caused the problem. Within our transformation framework, it is possible to fuse loops with different domains without prior peeling transformations because hoisting of control structures is delayed until code generation.

Pattern matching is not the only limitation to transformation composition. Consider the example of Figure 3.11 which adds two references to the original program, `A[1]` in statement *A* and `A[i]` in statement *C*. These references do not compromise the ability to fuse the three loops, as shown in Figure 3.12. Optimizers based on more advanced rewriting systems [Vis01] and most non-syntactic representations [Kel96, O'B98, LO04] will still peel an iteration of the first and last loops. However, peeling the last iteration of the first loop introduces two dependences that prevent fusion with the third loop: backward motion — flow dependence on `A[1]` — and forward motion — anti-dependence on `A[i]` — of the peeled iteration is now illegal. KAP yields the partially fused code in Figure 3.13, whereas our framework may still fuse the three loops as in Figure 3.12.

To address the composition issue, compilers come with an ordered set of phases. This approach is legitimate but prevents transformations to be applied several times, e.g., after some other transformation has modified the appropriateness of further optimizations. We consider again the example of Figure 3.5, and we now assume `A` is a local array only used to compute `B`. KAP applies dead-code elimination before fusion: it tries to eliminate `A`, but since it is used to compute `B`, it fails. Then the compiler fuses the two loops, and scalar promotion replaces `A` with a scalar, as shown in Figure 3.9. It is now obvious that array `A` can be eliminated but dead-code elimination will not be run again. Conversely, if we delayed dead-code elimination until after loop fusion (and peeling), we would still not fuse with the third loop but we would eliminate `A` as well as the peeled iteration, as shown in Figure 3.10. Clearly, both phase orderings lead to sub-optimal results. However, if we compile the code from Figure 3.9 with KAP — as if we applied the KAP sequence of transformations twice — array `A` and the peeled iteration are eliminated, allowing the compiler to fuse the three loops, eventually reaching the target optimized program of Figure 3.6.

These simple examples illustrate the artificial restrictions to transformation composition and the consequences on permuting or repeating transformations in current syntactic compilers. *Beyond parameter tuning, existing compilation infrastructures may not be very appropriate for iterative compilation.* By design, it is hard to modify either phase ordering or selection, and it is even harder to get any transformation pattern to match a significant part of the code after a long sequence of transformations.

3.1.2 Introduction to the Polyhedral Model

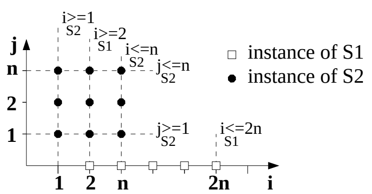
This section is a quick overview of the polyhedral framework; it also presents notations used throughout the chapter. A more formal presentation of the model may be found in [Pug91c, Fea92]. Polyhedral compilation usually distinguishes three steps: one first has to represent an input program in the formalism, then apply a transformation to this representation, and finally generate the target (syntactic) code.

Consider the polynomial multiplication kernel in Figure 3.14(a). It only deals with control aspects of the program, and we refer to the two computational statements (array assignments) through their names, S_1 and S_2. To bypass the limitations of syntactic representations, the polyhedral model is closer to the execution itself by considering *statement instances*. For each statement we consider the *iteration domain*, where every statement instance belongs. The domains are described using affine constraints that can be extracted from the program control. For example, the iteration domain of statement S_1, called $\mathcal{D}_{\mathrm{om}}^{S_1}$, is the set of values (i) such that $2 \le i \le n$ as shown in Figure 3.14(b); a matrix representation is used to represent such constraints: in our example, $\mathcal{D}_{\mathrm{om}}^{S_1}$ is characterized by

$$\begin{bmatrix} 1 & 0 & -2 \\ -1 & 2 & 0 \end{bmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} \ge \mathbf{0}.$$

```
     for (i=2; i<=2*n; i++)
S1   | Z[i] = 0;
     for (i=1; i<=n; i++)
     | for (j=1; j<=n; j++)
S2   | | Z[i+j] += X[i] * Y[j];
```

(a) Syntactic form



(b) Polyhedral domains ($n \ge 2$)

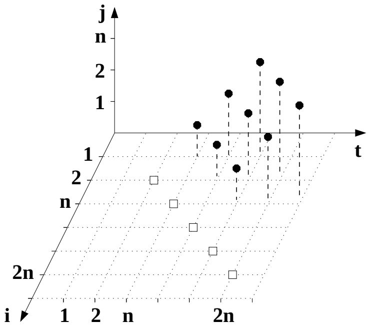
Figure 3.14: A polynomial multiplication kernel and its polyhedral domains

In this framework, a transformation is a set of *affine scheduling functions*. Each statement has its own scheduling function which maps each run-time statement instance to a logical execution date. In our polynomial multiplication example, an optimizer may notice a locality problem and discover a good data reuse potential over array `Z`, then suggest $\theta^{S_1}(i) = (i)$ and $\theta^{S_2}\begin{pmatrix} i \\ j \end{pmatrix} = (i + j + 1)$ to achieve better locality (see e.g., [Bas03] for a method to compute such functions). The intuition behind such transformation is to execute consecutively the instances of S_2

having the same $i+j$ value (thus accessing the same array element of Z) and to ensure that the initialization of each element is executed by S_1 just before the first instance of S_2 referring this element. In the polyhedral model, a transformation is applied following the template formula in Figure 3.15(a) [Bas04], where **i** is the iteration vector, $\mathbf{i}_{\mathrm{gp}}$ is the vector of constant parameters, and **t** is the *time-vector*, i.e. the vector of the scheduling dimensions. The next section will detail the nature of these vectors and the structure of the Θ and Λ matrices. Notice in this formula, equality constraints capture schedule modifications, and inequality constraints capture iteration domain modifications. The resulting polyhedra for our example are shown in Figure 3.15(b), with the additional dimension t.

$$\left(\begin{array}{c|c} \mathrm{I} & \Theta \\ \hline 0 & \Lambda \end{array}\right) \cdot \left(\begin{array}{c} -\mathbf{t} \\ \hline \mathbf{i} \\ \mathbf{i}_{\mathrm{gp}} \\ 1 \end{array}\right) \begin{array}{c} = \mathbf{0} \\ \hline \geq \mathbf{0} \end{array}$$

(a) Transformation template formula



(b) Transformed polyhedra

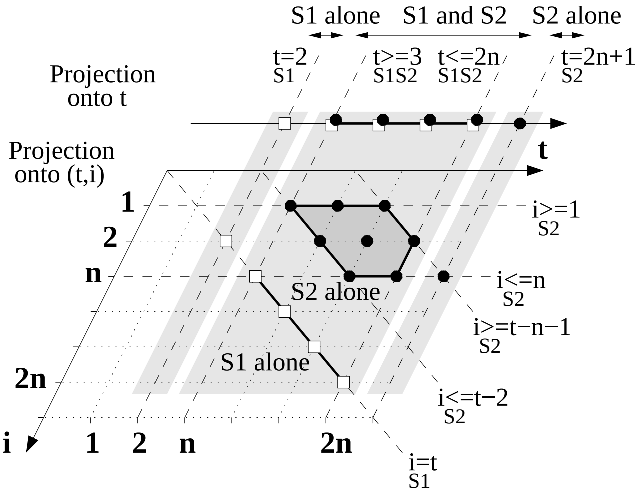
Figure 3.15: Transformation template and its application

Once transformations have been applied in the polyhedral model, one needs to (re)generate the target code. The best syntax tree construction scheme consists in a recursive application of domain projections and separations [QRW00, Bas04]. The final code is deduced from the set of constraints describing the polyhedra attached to each node in the tree. In our example, the first step is a projection onto the first dimension t, followed by a separation into disjoint polyhedra, as shown on the top of Figure 3.16(a). This builds the outer loops of the target code (the loops with iterator t in Figure 3.16(b)). The same process is applied onto the first two dimensions (bottom of Figure 3.16(a)) to build the second loop level and so on. The final code is shown in Figure 3.16(b) (the reader may care to verify that this solution maximally exploits temporal reuse of array Z). Note that the separation step for two polyhedra needs three operations: $\mathcal{D}_{\mathrm{om}}^{S_1} - \mathcal{D}_{\mathrm{om}}^{S_2}$, $\mathcal{D}_{\mathrm{om}}^{S_2} - \mathcal{D}_{\mathrm{om}}^{S_1}$ and $\mathcal{D}_{\mathrm{om}}^{S_2} \cap \mathcal{D}_{\mathrm{om}}^{S_1}$, thus for n statements the worst-case complexity is 3^n.

It is interesting to note that the target code, although obtained after only one transformation step, is quite different from the original loop nest. Indeed, multiple classical loop transformations are be necessary to simulate this one-step optimization (among them, software pipelining and skewing). The intuition is that arbitrarily complex compositions of classical transformations can be captured in one single transformation step of the polyhedral model. This was best illustrated by affine scheduling [Fea92, Kel96] and partitioning [LL97] algorithms. Yet, because black-box, model-based optimizers fail on modern processors, we propose to step back a little bit *and consider again the benefits of composing classical loop transformations, but using a polyhedral representation*. Indeed, up to now, polyhedral optimization frameworks have only considered the isolated application of one arbitrarily complex affine transformation. The main originality of our work is to address the *composition of program transformations on the polyhedral representation itself.* The next section presents the main ideas allowing to define compositions of affine transformations without intermediate code generation steps.

3.1.3 Isolating Transformations Effects

Let us now explain how our framework can separately and independently represent the iteration domain, the statements schedule, the data layout and the access functions of array references. At the same time, we will outline why this representation has several benefits for the implementation of program transformations: (1) it is generic and can serve to implement a large array of program transformations, (2) it isolates the root effects of program transformations, (3) it allows generalized versions of classical loop transformations to be defined without reference to any syntactic code, (4) this enables transparent composition of program transformations because applying program transformations has no effect on the representation complexity that makes it less generic or harder to manipulate, (5) and this eventually adds structure (commutativity, confluence, linearity) to the optimization search space.



(a) Projections an separations

```
    t=2; // Such equality is a loop running once
    | i=2;
S1  | | Z[i] = 0;
    for (t=3; t<=2*n; t++)
    | for (i=max(1,t-n-1); i<=min(t-2,n); i++)
    | | j = t-i-1;
S2  | | Z[i+j] += X[i] * Y[j]
    | i=t;
S1  | | Z[i] = 0;
    t=2*n+1;
    | i=n;
    | | j=n;
S2  | | | Z[i+j] += X[i] * Y[j];
```

(b) Target code

Figure 3.16: Target code generation

Principles

The scope of our representation is a sequence of loop nests with constant strides and affine bounds. It includes non-rectangular loops, non-perfectly nested loops, and conditionals with boolean expressions of affine inequalities. Loop nests fulfilling these hypotheses are amenable to a representation in the polyhedral model [PD96]. We call *Static Control Part* (SCoP) any *maximal syntactic program segment* satisfying these constraints [CGT04]. We only describe analyses and transformations confined within a given SCoP; the reader interested in techniques to extend SCoP coverage (by preliminary transformations) or in partial solutions on how to remove this scoping limitation (procedure abstractions, irregular control structures, etc.) should refer to [TFJ86, GC95, Col95, Won95, Cre96, BCF97, RP99, BCC00, Coh99, Col02].

All variables that are invariant within a SCoP are called *global parameters*; e.g., M, N, P and Q are the global parameters of the introductory example (see Figure 3.1). For each statement within a SCoP, the representation separates four attributes, characterized by parameter matrices: the iteration domain, the schedule, the data layout and the access functions. Even though transformations can still be applied to loops or full procedures, they are individually applied to each statement.

Iteration domains

Strip-mining and loop unrolling only modify the iteration domain — the number of loops or the loop bounds — but they do not affect the order in which statement instances are executed (the program schedule) or the way arrays are accessed (the memory access functions). To isolate the effect of such transformations, we define a representation of the iteration domain.

Although the introductory example contains 4 loops, i, j, k and l, S_2 and S_3 have a different two-dimensional iteration domain. Let us consider the iteration domain of statement S_2; it is defined as follows: $\{(i,j) \mid 0 \le i, i \le M-1, 0 \le j, j \le N-1\}$. The iteration domain matrix has one column for each iterator and each global parameter, here respectively i, j and M, N, P, Q. Therefore, the actual matrix representation of statement S_2 is

$$\begin{array}{rr|cccc|r|l}
i & j & M & N & P & Q & 1 & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \le i \\
-1 & 0 & 1 & 0 & 0 & 0 & -1 & i \le M-1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \le j \\
0 & -1 & 0 & 1 & 0 & 0 & -1 & j \le N-1
\end{array}$$

Example: implementing strip-mining All program transformations that only modify the iteration domain can now be expressed as a set of elementary operations on matrices (adding/removing rows/columns, and/or modifying the values of matrix parameters). For instance, let us strip-mine loop j by a factor B (a statically known integer), and let us consider the impact of this operation on the representation of the iteration domain of statement S_2.

Two loop modifications are performed: loop jj is inserted before loop j and has a stride of B. In our representation, loop j can be described by the following iteration domain inequalities: $jj \le j, j \le jj + B - 1$. For the non-unit stride B of loop jj, we introduce *local variables* to keep a linear representation of the iteration domain. For instance, the strip-mined iteration domain of S_2 is $\{(i,jj,j) \mid 0 \le j, j \le N-1, jj \le j, j \le jj+B-1, jj \bmod B = 0, 0 \le i, i \le M-1\}$, and after introducing local variable jj_2 such that $jj = B \times jj_2$, the iteration domain becomes $\{(i,jj,j) \mid \exists jj_2, 0 \le j, j \le N-1, jj \le j, j \le jj+B-1, jj = B \times jj_2, 0 \le i, i \le M-1\}$[2] and its matrix representation is the following (with $B = 64$, and from left to right: columns i, jj, j, jj_2, M, N, P, Q and the affine component):

$$\begin{array}{rrr|r|cccc|r|l}
i & j & jj & jj_2 & M & N & P & Q & 1 & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \le i \\
-1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & i \le M-1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \le j \\
0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & -1 & j \le N-1 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & jj \le j \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 63 & j \le jj+63 \\
0 & -1 & 0 & 64 & 0 & 0 & 0 & 0 & 0 & jj \le 64 \times jj_2 \\
0 & 1 & 0 & -64 & 0 & 0 & 0 & 0 & 0 & 64 \times jj_2 \le jj
\end{array}$$

Notations and formal definition Given a statement S within a SCoP, let d_S be the depth of S, $\mathbf{i}$ the vector of loop indices to which S belongs (the dimension of $\mathbf{i}$ is d_S), $\mathbf{i}_{\text{lv}}$ the vector of d_{lv} local variables added to linearize constraints, $\mathbf{i}_{\text{gp}}$ the vector of d_{gp} global parameters, and Λ^S the matrix of n linear constraints (Λ^S has n rows and $d^S + d^S_{\text{lv}} + d_{\text{gp}} + 1$ columns). The iteration domain of S is defined by

$$\mathcal{D}^S_{\text{om}} = \left\{\mathbf{i} \mid \exists \mathbf{i}_{\text{lv}}, \Lambda^S \times \left[\mathbf{i}, \mathbf{i}_{\text{lv}}, \mathbf{i}_{\text{gp}}, 1\right]^t \ge \mathbf{0}\right\}.$$

Schedules

Feautrier [Fea92], Kelly and Pugh [Kel96], proposed an encoding that characterizes the order of execution of each statement instance within code sections with multiple and non-perfectly nested loop nests. We use a similar encoding for SCoPs. The principle is to define a *time stamp* for each statement instance, using the iteration vector of the surrounding loops, e.g., vector (i,j) for statement S_2 in the introductory example, and the static statement order to accommodate loop levels with multiple statements. This statement order is defined for each loop level and starts to 0, e.g., the rank of statement S_2 is 1 at depth 1 (it belongs to loop j which is the second statement at depth 1 in this SCoP), 0 at depth 2 (it is the first statement in loop j). And for each statement, the encoding defines a schedule matrix Θ that characterizes the schedule. E.g., the instance (i,j) of statement S_2 is executed before the instance (k,l) of statement S_3 if and only if

$$\Theta^{S_2} \times \left[i,j,1\right]^t \ll \Theta^{S_3} \times \left[k,l,1\right]^t$$

(the last component in the instance vector $(i,j,1)$ — term 1 — is used for the static statement ordering term). Matrix Θ^{S_2} is shown in Figure 3.17, where the first two columns correspond to i,j and the last column corresponds to the static statement order. The rows of Θ^{S_2} interleave statement order and iteration order so as to implement

[2] The equation $jj = B \times jj_2$ is simply represented by two inequalities $jj \ge B \times jj_2$ and $jj \le B \times jj_2$.

the lexicographic order: the first row corresponds to depth 0, the second row to the iteration order of loop i, the third row to the static statement order within loop i, the fourth row to the iteration order of loop j, and the fifth row to the static statement order within loop j. Now, the matrix of statement Θ^{S_3} in Figure 3.17 corresponds to a different loop nest with different iterators.

$$\Theta^{S_2} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \qquad \Theta^{S_3} = \left[\begin{array}{cc|c} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \qquad \Theta^{S_2'} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \qquad \Theta^{S_2''} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Figure 3.17: Schedule matrix examples

Still, thanks to the lexicographic order, the encoding provides a global ordering, and we can check that $\Theta^{S_2} \times [i, j, 1] \ll \Theta^{S_3} \times [k, l, 1]$; in that case, the order is simply characterized by the static statement order at depth 0.

Because the schedule relies on loop iterators, iteration domain modifications — such as introducing a new loop (e.g., strip-mining) — will change the Θ matrix of all loop statements but not the schedule itself. Moreover, adding/removing local variables has no impact on Θ.

We will later see that this global ordering of all statements enables the transparent application of complex transformations like loop fusion.

Formal definition Let A^S be the matrix operating on iteration vectors, d^S the depth of the statement and β^S the static statement ordering vector. The schedule matrix Θ^s is defined by

$$\Theta^S = \left[\begin{array}{ccc|c} 0 & \cdots & 0 & \beta_0^S \\ \mathrm{A}_{1,1}^S & \cdots & \mathrm{A}_{1,d^S}^S & 0 \\ 0 & \cdots & 0 & \beta_1^S \\ \mathrm{A}_{2,1}^S & \cdots & \mathrm{A}_{2,d^S}^S & 0 \\ \vdots & \ddots & \vdots & \vdots \\ \mathrm{A}_{d^S,1}^S & \cdots & \mathrm{A}_{d^S,d^S}^S & 0 \\ 0 & \cdots & 0 & \beta_{d^S}^S \end{array}\right]$$

Example: implementing loop interchange and tiling As for unimodular transformations, applying a schedule-only loop transformation like loop interchange simply consists in swapping two rows of matrix Θ, i.e., really two rows of matrix A. Consider loops i and j the introductory example; the new matrix for S_2 associated with the interchange of i and j is called $\Theta^{S_2'}$ in Figure 3.17.

Now, tiling is a combination of strip-mining and loop interchange and it involves both an iteration domain and a schedule transformation. In our split representation, tiling loop j by a factor B simply consists in applying the iteration domain transformation in the previous paragraph (see the strip-mining example) and the above schedule transformation on all statements within loops i and j. For statement S_2, the only difference with the above loop interchange example is that strip-mining introduces a new loop iterator jj. The transformed matrix is called $\Theta^{S_2''}$ in Figure 3.17.

Extending the representation to implement more transformations For some statement-wise transformations like shifting (or pipelining), i.e., loop shifting for one statement in the loop body but not the others (e.g., statements S_2 and S_3, after merging loops i, k and j, l), more complex manipulations of the statement schedule are necessary. In fact, the above schedule representation is a simplified version of the actual schedule which includes a third matrix component called Γ. It adds one column to the Θ matrix for every global parameter (e.g., 4 columns for the running example).

Access functions

Privatization modifies array accesses, i.e., array subscripts. For any array reference, a given point in the iteration domain is mapped to an array element (for scalars, all iteration points naturally map to the same element). In other words, there is a function that maps the iteration domain of any reference to array or scalar elements. A transformation like privatization modifies this function: it affects neither the iteration domains nor the schedules.

Consider array reference `B[j][i]`, in statement S_2 after merging loops i, k and j, l, and strip-mining loop j. The matrix for the corresponding access function is simply (columns are i,jj, j,M,N,P,Q, and the scalar component, from left to right):

$$\left[\begin{array}{ccc|cccc|c} 0&0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0&0 \end{array}\right].$$

Formal definition For each statement S, we define two sets $\mathcal{L}^S_{\rm hs}$ and $\mathcal{R}^S_{\rm hs}$ of $(\mathtt{A}, f)$ pairs, each pair representing a reference to variable `A` in the left or right hand side of the statement; f is the *access function* mapping iterations in $\mathcal{D}^S_{\rm om}$ to `A` elements. f is a function of loop iterators, local variables and global parameters. The access function f is defined by a matrix F such that

$$f(\mathbf{i}) = \mathrm{F} \times \left[\mathbf{i}, \mathbf{i}_{\rm lv}, \mathbf{i}_{\rm gp}, 1\right]^t.$$

Example: implementing privatization Consider again the example in Figure 3.1 and assume that, instead of splitting statement `Z[i]=0` to enable tiling, we want to privatize array `Z` over dimension j (as an alternative). Besides modifying the declaration of `Z` (see next section), we need to change the subscripts of references to `Z`, adding a row to each access matrix with a 1 in the column corresponding to the new dimension and zeroes elsewhere. E.g., privatization of $\mathcal{L}^{S_2}_{\rm hs}$ yields

$$\left\{\left(\mathtt{Z}, \left[\begin{array}{cc|cccc|c} 1&0&0&0&0&0&0 \end{array}\right]\right)\right\} \longrightarrow \left\{\left(\mathtt{Z}, \left[\begin{array}{cc|cccc|c} 1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0 \end{array}\right]\right)\right\}.$$

Data layout

Some program transformations, like padding, only modify the array declarations and have no impact on the polyhedral representation of statements. It is critical to define these transformations through a separate representation of the mapping of virtual array elements to physical memory location. We do not improve on the existing solutions to this problem [O'B98], which are sufficiently mature already to express complex data layout transformations.

Notice a few program transformations can affect both array declarations and array statements. For instance, array merging (combining several arrays into a single one) affects both the declarations and access functions (subscripts change); this transformation is sometimes used to improve spatial locality. We are working on an extension of the representation to accommodate combined modifications of array declarations and statements, in the light of [O'B98]. This extension will revisit the split of the schedule matrix into independent parts with separated concerns, to facilitate the expression and the composition of data layout transformations. A similar split may be applicable to access functions as well.

3.1.4 Putting it All Together

Our representation allows us to compose transformations without reference to a syntactic form, as opposed to previous polyhedral models where a single-step transformation captures the whole loop nest optimization [Fea92, LL97] or intermediate code generation steps are needed [Wol92, Kel96].

Let us better illustrate the advantage of expressing loop transformations as "syntax-free" function compositions, considering again the example in Figure 3.1. The polyhedral representation of the original program is the following; statements are numbered S_1, S_2 and S_3, with global parameters $\mathbf{i}_{\rm gp} = [M,N,P,Q]^t$.

Statement iteration domains

$$\Lambda^{S_1} = \left[\begin{array}{c|cccc|c} 1&0&0&0&0&0\\ -1&1&0&0&0&-1 \end{array}\right] \begin{array}{l} 0\le i\\ i\le M-1 \end{array}$$

$$\Lambda^{S_2} = \left[\begin{array}{cc|cccc|c} 1&0&0&0&0&0&0\\ -1&0&1&0&0&0&-1\\ 0&1&0&0&0&0&0\\ 0&-1&0&1&0&0&-1 \end{array}\right] \begin{array}{l} 0\le i\\ i\le M-1\\ 0\le j\\ j\le N-1 \end{array} \quad \Lambda^{S_3} = \left[\begin{array}{cc|cccc|c} 1&0&0&0&0&0&0\\ -1&0&0&0&1&0&0\\ 0&1&0&0&0&0&0\\ 0&-1&0&0&0&1&0 \end{array}\right] \begin{array}{l} 0\le i\\ i\le P\\ 0\le j\\ j\le Q \end{array}$$

Statement schedules

$$
\begin{array}{lll}
\begin{array}{l}
A^{S_1} = [1] \\
\beta^{S_1} = [0\,0]^t \\
\Gamma^{S_1} = [0\,0\,0\,0 \mid 0]
\end{array}
&
\begin{array}{l}
A^{S_2} = \begin{bmatrix} 1\,0 \\ 0\,1 \end{bmatrix} \\
\beta^{S_2} = [0\,1\,0]^t \\
\Gamma^{S_2} = \left[\begin{array}{c|c} 0\,0\,0\,0 & 0 \\ 0\,0\,0\,0 & 0 \end{array}\right]
\end{array}
&
\begin{array}{l}
A^{S_3} = \begin{bmatrix} 1\,0 \\ 0\,1 \end{bmatrix} \\
\beta^{S_3} = [1\,1\,0]^t \\
\Gamma^{S_3} = \left[\begin{array}{c|c} 0\,0\,0\,0 & 0 \\ 0\,0\,0\,0 & 0 \end{array}\right]
\end{array}
\\
\text{i.e. } \Theta^{S_1} = \left[\begin{array}{c|c} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{array}\right]
&
\text{i.e. } \Theta^{S_2} = \left[\begin{array}{c|c} 0\,0 & 0 \\ 1\,0 & 0 \\ 0\,0 & 1 \\ 0\,1 & 0 \\ 0\,0 & 0 \end{array}\right]
&
\text{i.e. } \Theta^{S_3} = \left[\begin{array}{c|c} 0\,0 & 1 \\ 1\,0 & 0 \\ 0\,0 & 1 \\ 0\,1 & 0 \\ 0\,0 & 0 \end{array}\right]
\end{array}
$$

Statement access functions

$$
\begin{array}{l}
\mathcal{L}_{\text{hs}}^{S_1} = \Big\{ \Big(\mathsf{Z}, [1 | 0\,0\,0\,0 | 0] \Big) \Big\} \qquad \mathcal{R}_{\text{hs}}^{S_1} = \Big\{ \quad \Big\} \\
\mathcal{L}_{\text{hs}}^{S_2} = \Big\{ \Big(\mathsf{Z}, [1\,0 | 0\,0\,0\,0 | 0] \Big) \Big\} \\
\mathcal{R}_{\text{hs}}^{S_2} = \Big\{ \Big(\mathsf{Z}, [1\,0 | 0\,0\,0\,0 | 0] \Big), \Big(\mathsf{A}, \left[\begin{array}{c|c|c} 1\,0 & 0\,0\,0\,0 & 0 \\ 0\,1 & 0\,0\,0\,0 & 0 \end{array}\right] \Big), \Big(\mathsf{B}, \left[\begin{array}{c|c|c} 0\,1 & 0\,0\,0\,0 & 0 \\ 1\,0 & 0\,0\,0\,0 & 0 \end{array}\right] \Big), \Big(\mathsf{X}, [0\,1 | 0\,0\,0\,0 | 0] \Big) \Big\} \\
\mathcal{L}_{\text{hs}}^{S_3} = \Big\{ \Big(\mathsf{Z}, [1\,0 | 0\,0\,0\,0 | 0] \Big) \Big\} \\
\mathcal{R}_{\text{hs}}^{S_3} = \Big\{ \Big(\mathsf{Z}, [1\,0 | 0\,0\,0\,0 | 0] \Big), \Big(\mathsf{A}, \left[\begin{array}{c|c|c} 1\,0 & 0\,0\,0\,0 & 0 \\ 0\,1 & 0\,0\,0\,0 & 0 \end{array}\right] \Big), \Big(\mathsf{X}, [0\,1 | 0\,0\,0\,0 | 0] \Big) \Big\}
\end{array}
$$

Step1: merging loops *i* and *k* Within the representation, merging loops i and k only influences the schedule of statement S_3, i.e., Θ^{S_3}. No other part of the polyhedral program representation is affected. After merging, statement S_3 has the same static statement order at depth 0 as S_2, i.e., 0; its statement order at depth 1 becomes 2 instead of 1, i.e., it becomes the third statement of merged loop i.

$$\beta^{S_3} = [0\,2\,0]^t$$

Step2: merging loops *j* and *l* Thanks to the normalization rules on the polyhedral representation, performing the previous step does not require the generation of a fresh syntactic form to apply loop fusion again on internal loops j and l. Although Θ^{S_3} has been modified, its internal structure still exhibits all opportunities for further transformations. This is a strong improvement on previous polyhedral representations.

Again, internal fusion of loops j and l only modifies Θ^{S_3}. Its static statement order at depth 2 is now 1 instead of 0, i.e., it is the second statement of merged loop j.

$$\beta^{S_3} = [0\,1\,1]^t$$

Step3: fission The fission of the first loop to split-out statement `Z[i]=0` has an impact on Θ^{S_2} and Θ^{S_3} since their statement order at depth 0 is now 1 instead of 0 (`Z[i]=0` is now the new statement of order 0 at depth 0), while their statement order at depth 1 (loop i) is decreased by 1.

$$\beta^{S_2} = [1\,0\,0]^t \quad \beta^{S_3} = [1\,0\,1]^t$$

Step4: strip-mining *j* Strip-mining loop j only affects the iteration domains of statements S_2 and S_3: it adds a local variable and an iterator (and thus 2 matrix columns to Λ^{S_2} and Λ^{S_3}) plus 4 rows for the new inequalities. It also affects the structure of matrices Θ^{S_2} and Θ^{S_3} to take into account the new iterator, but it does not change the schedule. Λ^{S_2} is the same as the domain matrix for S_2' in Section 3.1.3, and the other matrices are:

$$\Lambda^{S_3} = \left[\begin{array}{rrr|r|cccc|r} 1 & 0 & 0 & 0 & 0&0&0&0 & 0 \\ -1 & 0 & 0 & 0 & 0&0&1&0 & -1 \\ 0 & 0 & 1 & 0 & 0&0&0&0 & 0 \\ 0 & 0 & -1 & 0 & 0&0&0&1 & -1 \\ 0 & -1 & 1 & 0 & 0&0&0&0 & 0 \\ 0 & 1 & -1 & 0 & 0&0&0&0 & 63 \\ 0 & -1 & 0 & 64 & 0&0&0&0 & 0 \\ 0 & 1 & 0 & -64 & 0&0&0&0 & 0 \end{array}\right] \begin{array}{l} 0 \leq i \\ i \leq P-1 \\ 0 \leq j \\ j \leq Q-1 \\ jj \leq j \\ j \leq jj+63 \\ jj \leq 64jj_2 \\ 64jj_2 \leq jj \end{array}$$

$$\mathrm{A}^{S_2} = \begin{bmatrix} 1&0&0 \\ 0&1&0 \\ 0&0&1 \end{bmatrix}, \beta^{S_2} = [1\,0\,0\,0]^t \text{ and } \mathrm{A}^{S_3} = \begin{bmatrix} 1&0&0 \\ 0&1&0 \\ 0&0&1 \end{bmatrix}, \beta^{S_3} = [1\,0\,0\,1]^t$$

Step5: strip-mining *i* Strip-mining *i* has exactly the same effect for loop *i* and modifies the statements S_2 and S_3 accordingly.

Step6: interchanging *i* **and** *j* As explained before, interchanging *i* and *j* simply consists in swapping the second and fourth row of matrices Θ^{S_2} and Θ^{S_3}, i.e., the rows of A^{S_2} and A^{S_3}

$$\mathrm{A}^{S_2} = \mathrm{A}^{S_3} = \begin{bmatrix} 1&0&0&0 \\ 0&0&1&0 \\ 0&1&0&0 \\ 0&0&0&1 \end{bmatrix} \longrightarrow \Theta^{S_2} = \left[\begin{array}{cccc|c} 0&0&0&0 & 1 \\ 1&0&0&0 & 0 \\ 0&0&0&0 & 0 \\ 0&0&1&0 & 0 \\ 0&0&0&0 & 0 \\ 0&1&0&0 & 0 \\ 0&0&0&0 & 0 \\ 0&0&0&1 & 0 \\ 0&0&0&0 & 0 \end{array}\right] \longrightarrow \Theta^{S_3} = \left[\begin{array}{cccc|c} 0&0&0&0 & 1 \\ 1&0&0&0 & 0 \\ 0&0&0&0 & 0 \\ 0&0&1&0 & 0 \\ 0&0&0&0 & 0 \\ 0&1&0&0 & 0 \\ 0&0&0&0 & 0 \\ 0&0&0&1 & 0 \\ 0&0&0&0 & 1 \end{array}\right]$$

Summary Overall, none of the transformations has increased the number of statements. Only transformations which add new loops and local variables increase the dimension of some statement matrices but they do not make the representation less generic or harder to use for compositions, since they enforce the normalization rules.

3.1.5 Normalization Rules

The separation between the domain, schedule, data layout and access functions attributes plays a major role in the compositionality of polyhedral transformations. Indeed, actions on different attributes compose in a trivial way, e.g., strip-mining (iteration domain), interchange (schedule) and padding (data layout). Nevertheless, the previous definitions do not, alone, guarantee good compositionality properties. To achieve our goal, we need to define additional normalization rules.

A given program can have multiple polyhedral representations. This is not harmless when the applicability of a transformation relies on the satisfaction of representation prerequisites. For example, it is possible to merge two statements in two loops only if these two statements are consecutive at the loops depth; e.g., assume the statement order of these two statements is respectively 0 and 2 instead of 0 and 1; the statement order (and thus the schedule) is the same but the statements are not consecutive and fusion seems impossible without prior transformations. Even worse, if the two statements have identical β vectors, fusion makes sense only if their schedules span disjoint time iterations, which in turn depends on both their A and Γ components, as well as their iteration domains. Without enforcing strong invariants to the representation, it is hopeless to define a program transformation uniquely from the matrices. *Normalizing* the representation after each transformation step is a critical contribution of our framework. It proceeds as follows.

Schedule matrix structure. Among many encodings, we choose to partition Θ into three components: matrices A (for iteration reordering) and Γ (iteration shifting), and vector β (statement reordering, fusion, fission), capturing different kinds of transformations. This avoids cross-pollution between statement and iteration reordering, removing expressiveness constraints on the combination of loop fusion with unimodular transformations and shifting. It allows to compose schedule transformations without a costly normalization to the Hermite normal form.

Sequentiality. This is the most important idea that structures the whole unified representation design. In brief, distinct statements, or identical statements in distinct iterations, cannot have the same time stamp. Technically, this rule is slightly stronger than that: we require that the A component of the schedule matrix is non-singular, that all statements have a different β vector, and that no β vector may be the prefix of another one.

This invariant brings two strong properties: (1) it suppresses scheduling ambiguities at code generation time, and (2) it guarantees that rule-compliant transformation of the schedule and will preserve sequentiality of the whole SCoP, independently of iteration domains. The first property is required to give the scheduling algorithm full control on the generated code. The second one is a great asset for separating the concerns when defining, applying or checking a transformation; domain and schedule are strictly independent, as much as modifications to A may ignore modifications to β and vice versa.

It is very important to understand that schedule sequentiality is in no way a limitation in the context of deeply and explicitly parallel architectures. First of all, parallel affine schedules are not the only way to express parallelism (in fact, they are mostly practical to describe bulk-synchronous parallelism), and in case they would be used to specify a partial ordering of statement instances, it is always possible to extend the schedule with “spatial” dimensions to make A invertible [DRV00].

Schedule density. Ensure that all statements at the same depth have a consecutive β ordering (no gap).

Domain density. Generation of efficient imperative code when scanning $\mathbb{Z}$-polyhedra (a.k.a. lattice polyhedra or linearly bounded lattices) is known to be a hard problem [DR94, Bas03]. Although not an absolute requirement, we try to define transformations that do not introduce local variables in the iteration domain. In particular, we will see in the next section that we use a different, less intuitive and more implicit definition of strip-mining to avoid the introduction of a local variable in the constraint matrix.

Domain parameters. Avoid redundant inequalities and try to reduce integer overflows in domain matrices Λ by normalizing each row.

3.2 Revisiting Classical Transformations

The purpose of this section is to review, with more detail, the formal definition of classical transformations in our compositional setting. Let us first define elementary operations called *constructors*. Constructors make no assumption about the representation invariants and may violate them.

Given a vector v and matrix M with $\dim(v)$ columns and at least i rows, $\mathsf{AddRow}(\mathrm{M},i,v)$ inserts a new row at position i in M and fills it with the value of vector v, $\mathsf{RemRow}(\mathrm{M},i)$ does the opposite transformation. $\mathsf{AddCol}(\mathrm{M},j,v)$ and $\mathsf{RemCol}(\mathrm{M},j)$ play similar roles for columns.

Moving a statement S forward or backward is a common operation: the constructor $\mathsf{Move}(P,Q,o)$ leaves all statements unchanged except those satisfying

$$\forall S \in \mathcal{S}_{\mathrm{cop}}, P \sqsubseteq \beta^S \wedge (Q \ll \beta^S \vee Q \sqsubseteq \beta^S) : \beta^S_{\dim(P)} \leftarrow \beta^S_{\dim(P)} + o,$$

where $u \sqsubseteq v$ denotes that u is a prefix of v, where P and Q are *statement ordering prefixes* s.t. $P \sqsubseteq Q$ defining respectively the context of the move and marking the initial time-stamp of statements to be moved, and where offset o is the value to be added to the component at depth $\dim(P)$ of any statement ordering vector β^S prefixed by P and following Q. If o is positive, $\mathsf{Move}(P,Q,o)$ inserts o free slots before all statements S preceded by the statement ordering prefix Q at the depth of P; respectively, if o is negative, $\mathsf{Move}(P,Q,o)$ deletes $-o$ slots.

3.2.1 Transformation Primitives

From the earlier constructors, we define transformation *primitives* to serve as building blocks for transformation sequences. These primitives do enforce the normalization rules. Figure 3.18 lists typical primitives affecting the polyhedral representation of a statement. $\mathbf{1}_k$ denotes the vector filled with zeros but element k set to 1, i.e., $(0,\ldots,0,1,0,\ldots,0)$; likewise, $\mathbf{1}_{i,j}$ denotes the matrix filled with zeros but element (i,j) set to 1.

Like the Move constructor, primitives do not directly operate on loops or statements, but target a collection of statements and polyhedra whose statement-ordering vectors share a common prefix P. There are no prerequisites on the program representation to the application and composition of primitives.

We also specified a number of optional *validity prerequisites* that conservatively check for the semantical soundness of the transformation, e.g., there are validity prerequisites to check that no dependence is violated by a unimodular or array contraction transformation. When exploring the space of possible transformation sequences, validity prerequisites avoid wasting time on corrupt transformations.

Fusion and Fission best illustrate the benefit of designing loop transformations at the abstract semantical level of our unified polyhedral representation. First of all, loop bounds are not an issue since the code generator will handle any overlapping of iteration domains. For the fission primitive, vector (P,o) prefixes all statements concerned by the fission; and parameter b indicates the position where statement delaying should occur. For the fusion primitive, vector $(P,o+1)$ prefixes all statements that should be interleaved with statements prefixed by (P,o). Eventually, notice that fusion followed by fission — with the appropriate value of b — leaves the SCoP unchanged.

The expressive power of the latter two transformations can be generalized through the very expressive Motion primitive. This transformation can displace a block of statements prefixed by P to a location identified by vector T, preserving the nesting depth of all statements and enforcing normalization rules. This transformation ressembles a polyhedral "cut-and-paste" operation that completely abstracts all details of the programs other than statement ordering in multidimensional time. This primitive uses an additional notation: $\text{pfx}(V,d)$ computes the sub-vector composed of the first d components of V.

Unimodular implements any unimodular transformation, extended to arbitrary iteration domains and loop nesting. U denotes a unimodular matrix. Notice the multiplication operates on both A *and* Γ, effectively updating the parametric shift along with skewing, reversal and interchange transformations, i.e., preserving the relative shift with respect to the time dimensions it was applied upon.

Shift implements a kind of hierarchical software pipelining on the source code. It is extended with parametric iteration shifts, e.g., to delay a statement by N iterations of one surrounding loop. Matrix M implements the parameterized shift of the affine schedule of a statement. M must have the same dimension as Γ.

CutDom constrains a domain with an additional inequality, in the form of a vector c with the same dimension as a row of matrix Λ.

Extend inserts a new intermediate loop level at depth ℓ, initially restricted to a single iteration. This new iterator will be used in following code transformations.

AddLocalVar insert a fresh local variable to the domain and to the access functions. This local variable is typically used by CutDom.

Privatize and Contract implement basic forms of array privatization and contraction, respectively, considering dimension ℓ of the array. Privatization needs an additional parameter s, the size of the additional dimension; s is required to update the array declaration (it cannot be inferred in general, some references may not be affine). These primitives are simple examples updating the data layout and array access functions.

This table is not complete (e.g., it lacks index-set splitting and data-layout transformations), but it demonstrates the expressiveness of the unified representation.

Syntax	Effect
Unimodular (P,U)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \mathrm{A}^S \leftarrow \mathrm{U.A}^S;\ \Gamma^S \leftarrow \mathrm{U}.\Gamma^S$
Shift (P,M)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \Gamma^S \leftarrow \Gamma^S + \mathrm{M}$
CutDom (P,c)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \Lambda^S \leftarrow \mathsf{AddRow}\big(\Lambda^S, 0, c/\gcd(c_1,\ldots,c_{d^S+d^S_{\text{lv}}+d_{\text{gp}}+1})\big)$
Extend (P,ℓ,c)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \begin{cases} d^S \leftarrow d^S+1;\ \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S,c,0); \\ \beta^S \leftarrow \mathsf{AddRow}(\beta^S,\ell,0); \\ \mathrm{A}^S \leftarrow \mathsf{AddRow}(\mathsf{AddCol}(\mathrm{A}^S,c,0),\ell,\mathbf{1}_\ell); \\ \Gamma^S \leftarrow \mathsf{AddRow}(\Gamma^S,\ell,0); \\ \forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F},\ell,0) \end{cases}$
AddLocalVar (P)	$\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, d^S_{\text{lv}} \leftarrow d^S_{\text{lv}}+1;\ \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S, d^S+1, 0);$ $\forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{AddCol}(\mathrm{F}, d^S+1, 0)$
Privatize (A,ℓ)	$\forall S \in \mathcal{S}_{\text{cop}}, \forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F},\ell,\mathbf{1}_\ell)$
Contract (A,ℓ)	$\forall S \in \mathcal{S}_{\text{cop}}, \forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S_{\text{hs}} \cup \mathcal{R}^S_{\text{hs}}, \mathrm{F} \leftarrow \mathsf{RemRow}(\mathrm{F},\ell)$
Fusion (P,o)	$b = \max\{\beta^S_{\dim(P)+1} \mid (P,o) \sqsubseteq \beta^S\}+1$ $\mathsf{Move}((P,o+1),(P,o+1),b);\ \mathsf{Move}(P,(P,o+1),-1)$
Fission (P,o,b)	$\mathsf{Move}(P,(P,o,b),1);\ \mathsf{Move}((P,o+1),(P,o+1),-b)$
Motion (P,T)	if $\dim(P)+1 = \dim(T)$ then $b = \max\{\beta^S_{\dim(P)} \mid P \sqsubseteq \beta^S\}+1$ else $b=1$ $\mathsf{Move}(\text{pfx}(T,\dim(T)-1),T,b)$ $\forall S \in \mathcal{S}_{\text{cop}} \mid P \sqsubseteq \beta^S, \beta^S \leftarrow \beta^S + T - \text{pfx}(P,\dim(T))$ $\mathsf{Move}(P,P,-1)$

Figure 3.18: Some classical transformation primitives

Primitives operate on program representation while maintaining the structure of the polyhedral components (the invariants). Despite their familiar names, the primitives' practical outcome on the program representation is widely extended compared to their syntactic counterparts. Indeed, transformation primitives like fusion or interchange apply to sets of statements that may be scattered and duplicated at many different locations in the generated code. In addition, these transformations are not proper *loop* transformations anymore, since they apply to sets of statement iterations that may have completely different domains and relative iteration schedules. For example, one may interchange the loops surrounding one statement in a loop body without modifying the schedule of other statements, and without distributing the loop first. Another example is the fusion of two loops with different domains without peeling any iteration.

Previous encodings of classical transformations in a polyhedral setting — most significantly [Wol92] and [Kel96] — use Presburger arithmetic as an expressive *operating* tool for implementing and validating transformations. In addition to operating on polytopes, our work *generalizes* loop transformations to more abstract *polyhedral domain* transformations, without explicitly relying on a nested loop structure with known bounds and array subscripts to define the transformation.

Instead of anchoring loop transformations to a syntactic form of the program, limitting ourselves to what can be expressed with an imperative semantics, we define higher level transformations on the polyhedral representation itself, abstracting away the overhead (versioning, duplication) and constraints of the code generation process (translation to an imperative semantics).

Syntax	Effect	Comments
$\textsc{Interchange}(P,o)$	$\forall S \in \mathcal{S}_{\mathrm{cop}} \mid P \sqsubseteq \beta^S,$ $\left\{\begin{array}{l} \mathrm{U} = \mathrm{I}_{d^S} - \mathbf{1}_{o,o} - \mathbf{1}_{o+1,o+1} + \mathbf{1}_{o,o+1} + \mathbf{1}_{o+1,o}; \\ \textsc{Unimodular}(\beta^S, \mathrm{U}) \end{array}\right.$	swap rows o and $o+1$
$\textsc{Skew}(P,\ell,c,s)$	$\forall S \in \mathcal{S}_{\mathrm{cop}} \mid P \sqsubseteq \beta^S,$ $\left\{\begin{array}{l} \mathrm{U} = \mathrm{I}_{d^S} + s \cdot \mathbf{1}_{\ell,c}; \\ \textsc{Unimodular}(\beta^S, \mathrm{U}) \end{array}\right.$	add the skew factor
$\textsc{Reverse}(P,o)$	$\forall S \in \mathcal{S}_{\mathrm{cop}} \mid P \sqsubseteq \beta^S,$ $\left\{\begin{array}{l} \mathrm{U} = \mathrm{I}_{d^S} - 2 \cdot \mathbf{1}_{o,o}; \\ \textsc{Unimodular}(\beta^S, \mathrm{U}) \end{array}\right.$	put a -1 in (o,o)
$\textsc{StripMine}(P,k)$	$\forall S \in \mathcal{S}_{\mathrm{cop}} \mid P \sqsubseteq \beta^S,$ $\left\{\begin{array}{l} c = \dim(P); \\ \textsc{Extend}(\beta^S, c, c); \\ u = d^S + d^S_{\mathrm{lv}} + d_{\mathrm{gp}} + 1; \\ \textsc{CutDom}(\beta^S, -k \cdot \mathbf{1}_c + (\mathrm{A}^S_{c+1}, \Gamma^S_{c+1})); \\ \textsc{CutDom}(\beta^S, k \cdot \mathbf{1}_c - (\mathrm{A}^S_{c+1}, \Gamma^S_{c+1}) + (k-1)\mathbf{1}_u) \end{array}\right.$	insert intermediate loop constant column $k \cdot \mathbf{i}_c \leq \mathbf{i}_{c+1}$ $\mathbf{i}_{c+1} \leq k \cdot \mathbf{i}_c + k - 1$
$\textsc{Tile}(P,o,k_1,k_2)$	$\forall S \in \mathcal{S}_{\mathrm{cop}} \mid (P,o) \sqsubseteq \beta^S,$ $\left\{\begin{array}{l} \textsc{StripMine}((P,o), k_2); \\ \textsc{StripMine}(P, k_1); \\ \textsc{Interchange}((P,0), \dim(P)) \end{array}\right.$	strip outer loop strip inner loop interchange

Figure 3.19: Composition of transformation primitives

Naturally, this higher-level framework is beneficial for transformation composition. Figure 3.19 composes primitives into typical transformations. INTERCHANGE swaps the roles of $\mathbf{i}_o$ and $\mathbf{i}_{o+1}$ in the schedule of the matching statements; it is a fine-grain extension of the classical interchange making no assumption about the shape of the iteration domain. SKEW and REVERSE define two well known unimodular transformations, with respectively the skew factor s with it's coordinates (ℓ, c), and the depth o of the iterator to be reversed. STRIPMINE introduces a new iterator to strip the schedule and iteration domain of all statements at the depth of P into intervals of length k (where k is a *statically known integer*). This transformation is a sequence of primitives and does not resort to the insertion of any additional local variable, see Figure 3.19. TILE extends the classical loop tiling at of the two nested loops at the depth of P, using $k \times k$ blocks, with arbitrary nesting and iteration domains. Tiling and strip-mining always operate on *time* dimensions, hence the propagation of a line from the schedule matrix (from A and Γ) into the iteration domain constraints; it is possible to tile the surrounding time dimensions of any collection of statements with unrelated iteration domains and schedules.

3.2.2 Implementing Loop Unrolling

In the context of code optimization, one of the most important transformations is loop unrolling. A naive implementation of unrolling with statement duplications may result in severe complexity overhead for further transformations and for the code generation algorithm (its separation algorithm is exponential in the number of statements, in the worst case). Instead of implementing loop unrolling in the intermediate representation of our framework, we delay it to the code generation phase and perform full loop unrolling in a *lazy* way. This strategy is fully implemented in the code generation phase and is triggered by annotations (carrying depth information) of the statements whose surrounding loops need to be unrolled; unrolling occurs in the separation algorithm of the code generator [Bas04] when all the statements being printed out are marked for unrolling at the current depth.

Practically, in most cases, loop unrolling by a factor *b* an be implemented as a combination of *strip-mining* (by a factor *b*) and *full unrolling* [Wol96]. Strip-mining itself may be implemented in several ways in a polyhedral setting. Following our earlier work in [CGP$^+$05] and calling *b* the strip-mining factor, we choose to model a strip-mined loop by dividing the iteration span of the outer loop by *b* instead of leaving the bounds unchanged and inserting a non-unit stride *b*, see Figure 3.20.

```
for(i=ℓ(x⃗); i<=u(x⃗); i++)
```

$\xrightarrow{\text{strip-mine}\,(b)}$

```
for(t1=⌊ℓ(x⃗)/b⌋; t1<=⌊u(x⃗)/b⌋; t1++)
  for(t2=max(ℓ(x⃗),b*t1); t2<=min(u(x⃗),b*t1+b-1); t2++)
```

Figure 3.20: Generic strip-mined loop after code generation

This particular design preserves the convexity of the polyhedra representing the transformed code, alleviating the need for specific stride recognition mechanisms (based, e.g., on the Hermite normal form).

In Figure 3.21(b) we can see how strip-mining the original code of Figure 3.21(a) by a factor of 2 yields an internal loop with non-trivial bounds. It can be very useful to unroll the innermost loop to exhibit register reuse (a.k.a. register tiling), relax scheduling constraints and diminish the impact of control on useful code. However, unrolling requires to cut the domains so that `min` and `max` constraints disappear from loop bounds. Our method is presented in more detail in [VBC06]; it intuitively boils down to finding conditionals (lower bound and upper bound) *such that their difference is a non-parametric constant*: the unrolling factor. Hoisting these conditionals actually amounts to splitting the outer strip-mined loop into a kernel part where the inner strip-mined loop will be fully unrolled, and a remainder part (not unrollable) spanning at most as many iterations as the strip-mining factor. In our example, the conditions associated with a constant trip-count (equal to 2) are `t2>=2*t1` and `t2<=2*t1+1` and are associated with the kernel, separated from the prologue where `2*t1<M` and from the epilogue where `2*t1+1>N`. This separation leads to the more desirable form of Figure 3.21(c).

Finally, instead of implementing loop unrolling in the intermediate representation of our framework, we delay it to the code generation phase and perform full loop unrolling in a lazy way, avoiding the added (exponential) complexity on the separation algorithm. This approach relies on a preliminary strip-mine step that determines the amount of partial unrolling.

```
for(t1=M; t1<=N; t1++)
  S1(i = t1);
```

(a) Original code

```
for(t1=M/2; t1<=N/2; t1++)
  for(t2=max(M,2*t1);
       t2<=min(N,2*t1+1); t2++)
    S1(i = t2);
```

(b) Strip-mining of 2

```
if(M%2==1)
  S1(i = M);
for(t1=(M+1)/2; t1<=(N-1)/2; t1++)
  S1(i = 2*t1);
  S1(i = 2*t1+1);
if(N%2==0)
  S1(i = N);
```

(c) Separation & unrolling

Figure 3.21: Strip-mining and unrolling transformation

3.2.3 Parallelizing Transformations

Most parallelizing compilers rely on loop transformations to extract and expose parallelism, from vector and instruction-level to thread-level forms of parallelism [AK87, CHH$^+$93, BEF$^+$96, H$^+$96, KAP, CDS96, BGGT02, Nai04, EWO04]. The most common strategy is to compose loop transformations to extract parallel (`doall`) or pipeline (`doacross`) loops [BEF$^+$96]. The main transformations include privatization [MAL93, TP93, RP99] for dependence removal and unimodular transformations or node splitting to rearrange dependences [Ban88, Wol96].

Many academic approaches to automatic parallelization have used the polyhedral model — and partially ordered affine schedules in particular — to describe fine grain vector [Pug91a, Fea92, Xue94] or systolic [GQQ$^+$01, SAR$^+$00] parallelism. Affine schedules have also been applied to the extraction and characterization of bulk-synchronous parallelism [LL97, DRV00, LLL01]. Array expansion is a generalization of privatization that leverages on the precision of array dependence analysis in the polyhedral model [Fea88a, BCC98, BCC00]. Array contraction [Wol96, LLL01] and its generalization called storage mapping optimization [LF98, SCFS98, QR99] allows to control the overhead due to expansion techniques.

Our work does not aim at characterizing parallel execution with partially ordered affine schedules. In this sense, we prefer the more general and decoupled approach followed by traditional parallelizing compilers where parallelism is a separate concern. Loop transformations expressed on the schedule parts of the representation are seen as *enabling* transformations to extract parallel loops or independent instructions in loop bodies. These enabling transformations are associated with a precise dependence analysis to effectively allow to generate code with parallel execution annotations, using e.g., OpenMP. Modeling more dynamic forms of parallelism also requires such a decoupled approach: recently, a modernized version of Polaris has been extended to automatically extract vast amounts of effectively exploitable parallelism in irregular scientific codes, using *hybrid analysis* [RRH03], coupling a symbolic dependence analysis with an inference algorithm to guard parallel code with low-overhead dynamic tests wherever a fully static decision is not feasible. Yet these results used no prior loop transformation to enhance scalability through additional parallelism extraction or to coarsen its grain. Although we cannot show any empirical evidence yet, we believe the same reason why our framework improves on single-threaded optimizations (flexibility to express complex transformation sequences) will bring more scalability and robustness to these promising hybrid parallelization techniques.

3.2.4 Facilitating the Search for Compositions

To conclude this section, we study how our polyhedral representation with normalization rules for compositionality can further facilitate the search for complex transformation sequences.

We have seen that applying a program transformation simply amounts to recomputing the matrices of a few statements. This is a major increase in flexibility, compared to syntactic approaches where the code complexity increases with each transformation. It is still the case for prefetching and strip-mining, where, respectively, a statement is added and matrix dimensions are increased; but the added complexity is fairly moderate, and again the representation is no less generic.

Transformation Space

Commutativity properties are additional benefits of the separation into four representation aspects and the normalization rules. In general, data and control transformations commute, as well as statement reordering and iteration reordering. For example, loop fusion commutes with loop interchange, statement reordering, loop fission and loop fusion itself. In the example detailed in Section 3.1.4, swapping fusion and fission has no effect on the resulting representation; the first row of β vectors below shows double fusion followed by fission, while the second row shows fission followed by double fusion.

$$\begin{tikzcd}
\begin{array}{l}\beta^{S_1}=[0\,0]\\ \beta^{S_2}=[0\,1\,0]\\ \beta^{S_3}=[1\,0\,0]\end{array} \arrow[r] \arrow[d] & \begin{array}{l}\beta^{S_1}=[0\,0]\\ \beta^{S_2}=[0\,1\,0]\\ \beta^{S_3}=[0\,2\,0]\end{array} \arrow[r] & \begin{array}{l}\beta^{S_1}=[0\,0]\\ \beta^{S_2}=[0\,1\,0]\\ \beta^{S_3}=[0\,1\,1]\end{array} \arrow[d] \\
\begin{array}{l}\beta^{S_1}=[0\,0]\\ \beta^{S_2}=[1\,0\,0]\\ \beta^{S_3}=[2\,0\,0]\end{array} \arrow[r] & \begin{array}{l}\beta^{S_1}=[0\,0]\\ \beta^{S_2}=[1\,0\,0]\\ \beta^{S_3}=[1\,1\,0]\end{array} \arrow[r] & \begin{array}{l}\beta^{S_1}=[0\,0]\\ \beta^{S_2}=[0\,1\,0]\\ \beta^{S_3}=[0\,1\,1]\end{array}
\end{tikzcd}$$

Confluence properties are also available: outer loop unrolling and fusion (unroll-and-jam) is strictly equivalent to strip-mining, interchange and full unrolling. The latter sequence is the best way to implement unroll-and-jam in our framework, since it does not require statement duplication in the representation itself but relies on lazy unrolling. In general, strip-mining leads to confluent paths when combined with fusion or fission.

Such properties are useful in the context of iterative searches because they may significantly reduce the search space, and they also improve the understanding of its structure, which in turn enables more efficient search strategies [CST02].

Strip-mining and shifting do *not* commute. However applying shifting after strip-mining amounts to intra-tile pipelining (the last iteration of a tile stays in that tile), whereas the whole iteration space is pipelined across tiles when applying strip-mining after shifting (the last iteration of a tile being shifted towards the first iteration of the next tile).

When changing a sequence of transformations simply means changing a parameter

Finally, the code representation framework also opens up a new approach for searching compositions of program transformations. Since many program transformations have the only effect of modifying the matrix parameters, an alternative is to *directly search the matrix parameters themselves*. In some cases, changing one or a few parameters is equivalent to performing a sequence of program transformations, making this search much simpler and more systematic.

For instance, consider the Θ^{S_3} matrix of Section 3.1.3 and now assume we want to systematically search schedule-oriented transformations. A straightforward approach is to systematically search the Θ^{S_3} matrix parameters themselves. Let us assume that, during the search we randomly reach the following matrix:

$$\Theta^{S'_3} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right]$$

This matrix has 7 differences with the original Θ^{S_3} matrix of Section 3.1.3, and these differences actually correspond to the composition of 3 transformations: loop interchange (loops k and l), outer loop fusion (loops i and k) and inner loop fusion (loops j and l). In other words, searching the matrix parameters is equivalent to searching for compositions of transformations.

Furthermore, assuming that a full polyhedral dependence graph has been computed,[3] it is possible to characterize the *exact set of all schedule, domain and access matrices associated with legal transformation sequences*. This can be used to quickly filter out or correct any violating transformation [BF04], or even better, using the Farkas lemma as proposed by Feautrier [Fea92], to recast this implicit characterization into an explicit list of domains (of Farkas multipliers) enclosing the very values of all matrix coefficients associated with legal transformations. Searching for a proper transformation within this domain would be amenable to mathematical tools, like linear programming, promising better scalability than genetic algorithms on plain transformation sequences. This idea is derived from the “chunking” transformation for automatic locality optimization [BF03, BF04]; it is the subject of active ongoing work.

3.3 Higher Performance Requires Composition

We have already illustrated the need for long sequences of composed transformations and the limitations of syntactic approaches on the synthetic example of Section 3.1.1. As stated in the first chapter, empirical evidence on realistic benchmarks was provided by a methodological work by Parello et al. [PTCV04]. We only recall the main experimental results to enable further analysis of the requirements of a transformation composition framework.

3.3.1 Manual Optimization Results

Experiments were conducted on an HP AlphaServer ES45, 1GHz Alpha 21264C EV68 (1 processor enabled) with 8MB L2 cache and 8GB of memory. We compare our optimized versions with the *base* SPEC performance, i.e., the output of the HP Fortran (V5.4) and C (V6.4) compiler (`-arch ev6 -fast -O5 ONESTEP`) using the KAP Fortran preprocessor (V4.3).

[3] Our tool performs on-demand computation, with lists of polyhedra capturing the (exact) instance-wise dependence information between pairs of references.

Transformation sequences

In the following, we assume an aggressive inlining of all procedure calls within loops (performed by KAP in most cases). The examples in Figures 3.25, 3.23, 3.24 and 3.22 show a wide variability in transformation sequences and ordering. Each analysis and transformation phase is depicted as a gray box, showing the time difference when executing the *full benchmark* (in seconds, a negative number is a performance improvement); the base execution time for each benchmark is also indicated in the caption. Each transformation phase, i.e., each gray box, is then broken down into traditional transformations, i.e., white boxes.

All benchmarks benefited from complex compositions of transformations, with up to 23 individual loop and array transformations on the same loop nest for galgel. Notice that some enabling transformations actually degrade performance, like (A2) in galgel.

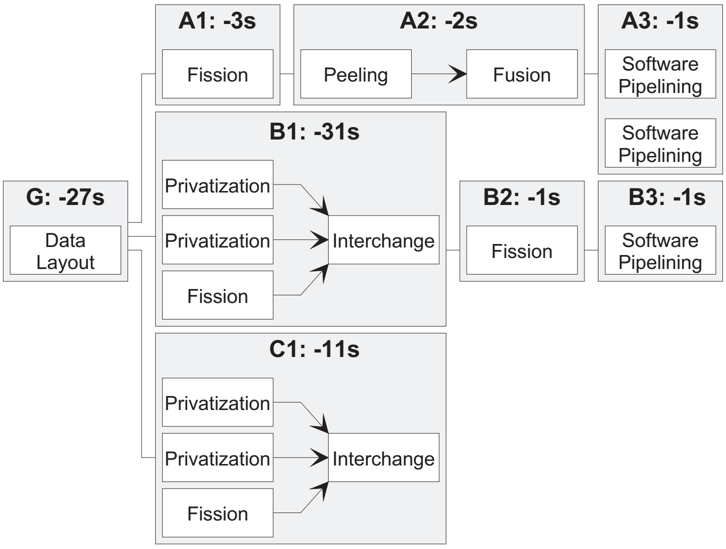


Figure 3.22: Optimizing apsi (base 378s)

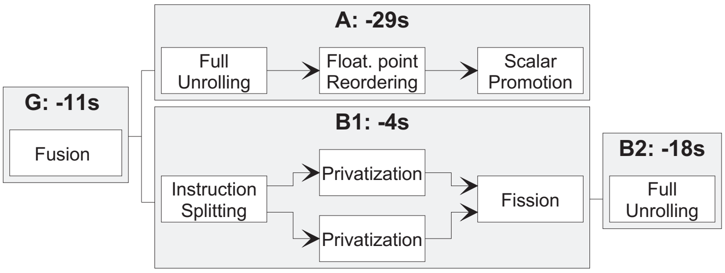


Figure 3.23: Optimizing applu (base 214s)

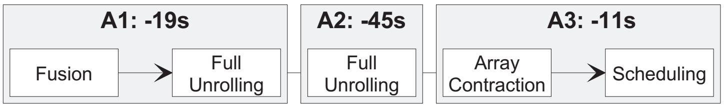


Figure 3.24: Optimizing wupwise (base 236s)

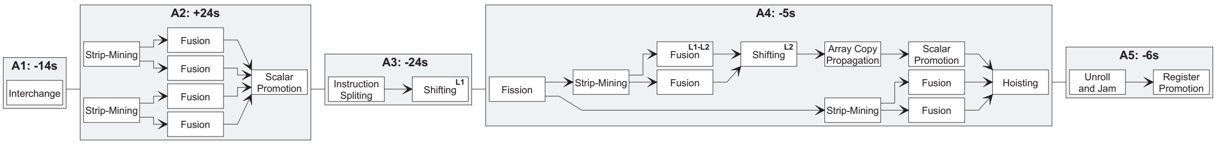


Figure 3.25: Optimizing galgel (base 171s)

3.3.2 Polyhedral vs. Syntactic Representations

Section 3.1 presented the main assets of our new polyhedral representation. We now revisit these properties on the 4 chosen benchmarks.

Code size and complexity

The manual application of transformation sequences leads to a large code size increase, let aside the effect of function inlining. This is due to code duplication when unrolling loops, but also to iteration peeling and loop versioning when applying loop tiling and strip-mining. Typical cases are phases (A) in applu and (A2) wupwise (unrolling), and (A5) in galgel (unroll-and-jam).

In our framework, none of these transformations perform any statement duplication, only strip-mining has a slight impact on the size of domain matrices, as explained in Section 3.1.3. In general, the only duplication comes from parameter versioning and from intrinsicly code-bloating schedules resulting from intricate transformation sequences. This “moral” observation allows to blame the transformation sequence rather than the polyhedral transformation infrastructure, yet it does not provide an intuitive characterization of the “good” transformation sequences that do not yield code-bloating schedules; this is left for future work.

Interestingly, it is also possible to control the aggressiveness of the polyhedral code generator, focusing its code-duplicating optimizations to the hottest kernels only, yielding sub-optimal but very compact code in the rest of the program. Again, the design of practical heuristics to drive these technique is left for future work.

Breaking patterns

On the introductory example, we already outlined the difficulty to merge loops with different bounds and tile non-perfectly nested loops. Beyond non-matching loop bounds and non-perfect nests, loop fusion is also inhibited by loop peeling, loop shifting and versioning from previous phases. For example, galgel shows multiple instances of fusion and tiling transformations after peeling and shifting. KAP’s pattern-matching rules fail to recognize any opportunity for fusion or tiling on these examples.

Interestingly, syntactic transformations may also introduce some spurious array dependences that hamper further optimizations. For example, phase (A3) in galgel splits a complex statement with 8 array references, and shifts part of this statement forward by one iteration (software pipelining) of a loop L_1. Then, in one of the fusion boxes of phase (A4), we wish to merge L_1 with a subsequent loop L_2. Without additional care, this fusion would *break dependences*, corrupting the semantics of the code produced after (A3). Indeed, some values flow from the shifted statement in L_1 to iterations of L_2; merging the loops would consume these values before producing them. Syntactic approaches lead to a dead-end in this case; the only way to proceed is to undo the shifting step, increasing execution time by 24 seconds. Thanks to the commutation properties of our model, we can make the dependence between the loops compatible with fusion by shifting the loop L_2 forward by one iteration, before applying the fusion.

Flexible and complex compositions of transformations

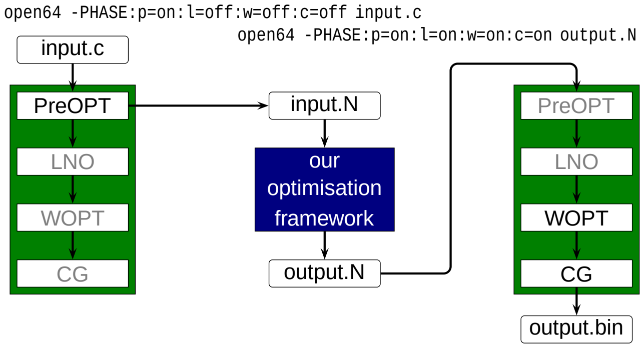
The manual benchmark optimizations exhibit wide variations in the composition of control, access and layout transformations. galgel is an extreme case where KAP does not succeed in optimizing the code, even with the best hand-tuned combination of switches, i.e., when directed to apply some transformations with explicit optimization switches (peak SPEC). Nevertheless, our (long) optimization sequence yields a significant speedup while only applying classical transformations. A closer look at the code shows only uniform dependences and constant loop bounds. In addition to the above-mentioned syntactic restrictions and pattern mismatches, our sequence of transformations shows the variability and complexity of enabling transformations. For example, to implement the eight loop fusions in Figure 3.25, strip-mining must be applied to convert large loops of N^2 iterations into nested loops of N iterations, allowing subsequent fusions with other loops of N iterations.

applu stresses another important flexibility issue. Optimizations on two independent code fragments follow an opposite direction: (G) and (A) target locality improvements: they implement loop fusion and scalar promotion; conversely, (B1) and (B2) follow a parallelism-enhancing strategy based on the opposite transformations: loop fission and privatization. Since the appropriate sequence is not the same in each case, the optimal strategy must be flexible enough to select either option.

Finally, any optimization strategy has an important impact on the order in which transformations are identified and applied. When optimizing applu and apsi, our methodology focused on individual transformations on separate loop nests. Only in the last step, dynamic analysis indicated that, to further improve performance, these loop nests must first be merged before applying performance-enhancing transformations. Of course, this is very much dependent on the strategy driving the optimization process, but an iterative feedback-directed approach is likely to be at least as demanding as a manual methodology, since it can potentially examine much longer transformation sequences.

3.4 Implementation

The whole infrastructure is implemented as a free (GPL) add-on to the Open64/ORC/EKOPath family of compilers [ORC, Cho04]. Optimization is performed in two runs of the compiler, with one intermediate run of our tool, using intermediate dumps of the intermediate representation (the `.N` files) as shown in Figure 3.26. It thus natively supports the generation of IA64 code. The whole infrastructure compiles with GCC3.4 and is compatible with PathScale EKOPath [Cho04] native code generator for AMD64 and IA32. Thanks to third-party tools based on Open64, this framework supports source-to-source optimization, using the robust C unparser of Berkeley UPC [BCBY04], and planning a port of the Fortran90 unparser from Open64/SL [CZT+]. It contains 3 main tools in addition to Open64: WRaP-IT which extracts SCoPs and build their polyhedral representation, URUK which performs program transformations in the polyhedral representation, and URGenT the code generator associated with the polyhedral representation[4].



```
open64 -PHASE:p=on:l=off:w=off:c=off input.c
open64 -PHASE:p=on:l=on:w=on:c=on output.N
```

Figure 3.26: Optimisation process

3.4.1 WRaP-IT: WHIRL Represented as Polyhedra — Interface Tool

WRaP-IT is an interface tool built on top of Open64 which converts the WHIRL — the compiler's hierarchical intermediate representation — to an augmented polyhedral representation, maintaining a correspondence between matrices in SCoP descriptions with the symbol table and syntax tree. Although WRaP-IT is still a prototype, it proved to be robust; the whole source-to-polyhedra-to-source conversion (without any intermediate loop transformation) was successfully applied in 34 seconds in average per benchmark on a 512MB 1GHz Pentium III machine.

Implemented within the modern infrastructure of Open64, WRaP-IT benefits from interprocedural analysis and pre-optimization phases such as function inlining, interprocedural constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, and induction variable substitution. Our tool

[4] These tools can be downloaded from `http://www.lri.fr/~girbal/site_wrapit`.

extracts large and representative SCoPs for SPECfp2000 benchmarks: on average, 88% of the statements belong to a SCoP containing at least one loop [BCG+03].

To refine these statistics, Figures 3.27 and 3.28 describe the SCoP breakdown for each benchmark with respect to instruction count and maximal loop nesting depth, respectively. These numbers confirm the lexical importance of code that can be represented in our model, and set a well defined scalability target for the (most of the time exponential) polyhedral computations associated with program analyses and transformations.

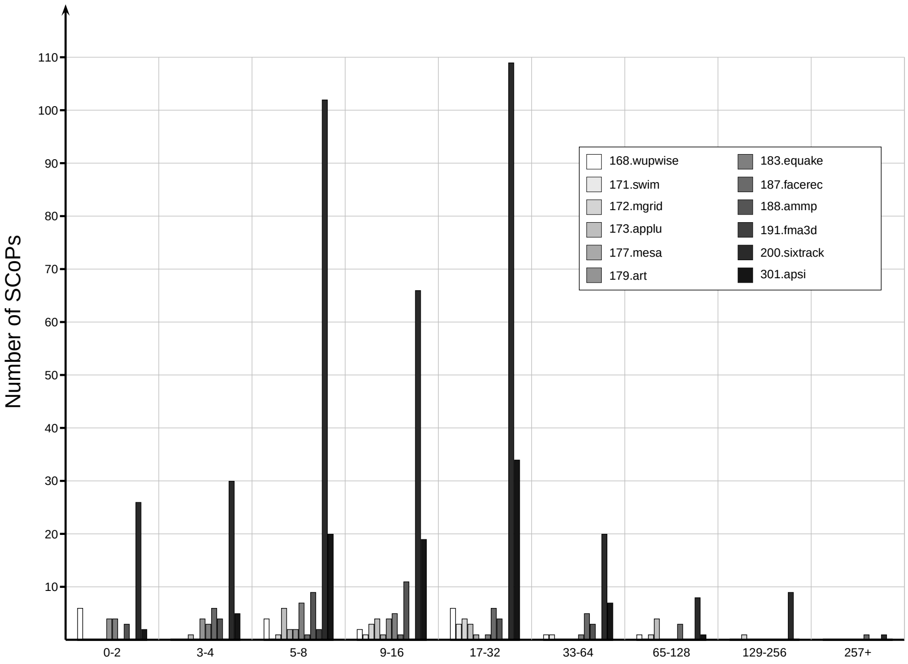


Figure 3.27: SCoP size (instructions)

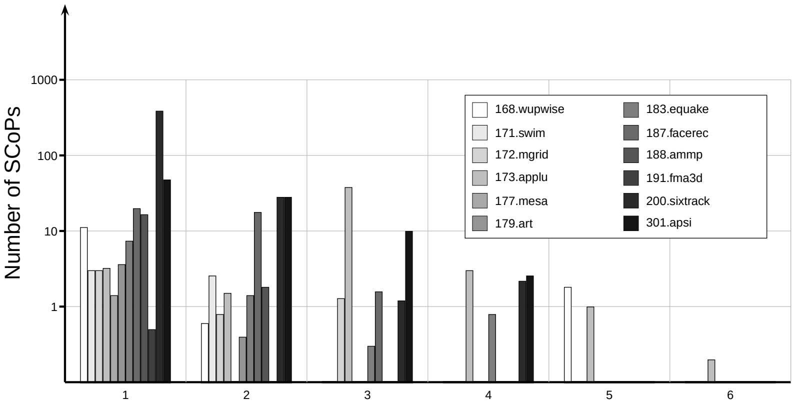


Figure 3.28: SCoP depth

To refine this coverage study, we computed the SCoP breakdown with respect to effective execution time. We conducted statistical sampling measurements, using the `oprofile` portable performance monitoring framework. Figure 3.29 gather the execution time percentage associated with each consecutive block of source statements (over 2.5% execution time). The penultimate column, #SCoPs, gives the number of SCoPs covering this code block: the lower the better. The last column shows the maximal loop nesting depth in those SCoPs and the actual loop nesting depth in the procedure; when the two numbers differ, some enclosing loops are not considered static control. In many cases, *a single full-depth* SCoP is sufficient to cover the whole block of “hot” instructions, showing that polyhedral transformations will be fully applicable to this code block. These results are very encouraging, yet far from sufficient in the context of general-purpose applications. This motivates further research in extending the applicability of polyhedral techniques to “sparsely irregular” code. Inlining was disabled to isolate SCoP coverage in each source code function.[5]

	File	Function	Source Lines	%Time	#SCoPs	SCoP Depth / Actual Depth
168.wupwise	zaxpy.f	zaxpy	11–32	20.6%	1	1/1
	zcopy.f	zcopy	11–24	8.3%	1	1/1
	zgemm.f	zgemm	236–271	47.5%	7	3/3
171.swim	swim.f	main	114–119	5.6%	1	2/2
	swim.f	calc1	261–269	26.3%	1	2/2
	swim.f	calc2	315–325	36.8%	1	2/2
	swim.f	calc3	397–405	29.2%	1	2/2
172.mgrid	mgrid.f	psinv	149–166	27.1%	1	3/3
	mgrid.f	resid	189–206	62.1%	1	3/3
	mgrid.f	rprj3	230–250	4.3%	1	3/3
	mgrid.f	interp	270–314	3.4%	1	3/3
173.applu	applu.f	blts	553–624	15.5%	1	6/6
	applu.f	buts	659–735	21.8%	1	6/6
	applu.f	jacld	1669–2013	17.3%	1	3/3
	applu.f	jacu	2088–2336	12.6%	1	3/3
	applu.f	rhs	2610–3068	20.2%	1	4/4
183.equake	quake.c	main	435–478	99%	4	2/3
187.facerec	cfftb.f90	passb4	266–310	35.6%	1	2/2
	gaborRoutines.f90	GaborTrafo	102–132	19.2%	2	2/2
	graphRoutines.f90	LocalMove	392–410	18.7%	2	0/4
	graphRoutines.f90	TopCostFct	451–616	8.23%	1	0/0
200.sixtrack	thin6d.f	thin6d	180–186	15.2%	1	1/3
	thin6d.f	thin6d	216–227	3.7%	1	1/3
	thin6d.f	thin6d	230–244	8.9%	3	1/3
	thin6d.f	thin6d	267–287	8.2%	2	1/3
	thin6d.f	thin6d	465–477	6.3%	1	1/3
	thin6d.f	thin6d	560–588	54.8%	1	2/4
301.apsi	apsi.f	dcdtz	1326–1354	4.3%	1	3/3
	apsi.f	dtdtz	1476–1499	4.3%	1	1/3
	apsi.f	dudtz	1637–1688	4.5%	1	3/3
	apsi.f	dvdtz	1779–1833	4.5%	1	3/3
	apsi.f	wcont	1878–1889	7.5%	1	1/3
	apsi.f	trid	3189–3205	5.9%	1	1/1
	apsi.f	smth	3443–3448	3.7%	1	1/1
	apsi.f	radb4	5295–5321	6.6%	2	2/2
	apsi.f	radbg	5453–5585	9.0%	3	3/3
	apsi.f	radf4	5912–5938	3.2%	2	2/2
	apsi.f	radfg	6189–6287	5.1%	2	3/3
	apsi.f	dkzmh	6407–6510	11.4%	8	1/3

Figure 3.29: Static and dynamic SCoP coverage

3.4.2 URUK: Unified Representation Universal Kernel

URUK is the key software component: it performs program transformations within the WRaP (polyhedral) representation. A scripting language, defines transformations and enables the composition of new transformations. Each transformation is built upon a set of elementary actions, the *constructors* (See Section 3.2).

Figure 3.30 shows the definition of the Move constructor, and Figure 3.31 defines the FISSION transformation based on this constructor. This syntax is preprocessed to overloaded C++ code, offering a high-level semantics to manipulate the polyhedral representation. It takes less than one hour for an URUK expert to implement a complex transformation like tiling of imperfectly nested loops with prolog/epilog generation and legality checks, and to have this transformation work on real benchmarks without errors.

Transformation composition is very natural in the URUK syntax. Figure 3.32 shows how simple it is to implement tiling from the composition of strip-mining and interchange primitives, hiding all the details associated with remainder loop management and legality checking.

3.4.3 URDeps: URUK Dependence Analysis

An important feature of URUK is the ability to perform transformations *without mandatory intermediate validity checks, and without reference to the syntactic program.* This allows to compute dependence information and to perform validity checks on demand. Our dependence analysis computes an *exact* information whenever possible, i.e., whenever array references are affine (control structures are assumed affine in SCoPs). A list of convex polyhedra is computed for each pair of statements and for each depth, *considering the polyhedral representation*

[5] We left out 6 SPECfp2000 benchmarks due to the (current) lack of support in our analyzer for function pointers and pointer arithmetic.

```
%transformation move
%param BetaPrefix P, Q
%param Offset o
%prereq P<=Q
%code
{
  foreach S in SCoP
    if (P<=S.Beta && Q<=S.Beta)
      S.Beta(P.dim())+=o;
    else if (P<=S.Beta && Q<<S.Beta)
      S.Beta(P.dim())+=o;
}
```

Figure 3.30: Move constructor

```
%transformation fission
%param BetaPrefix P
%param Offset o, b
%code
{
  UrukVector Q=P;
  Q.enqueue(o); Q.enqueue(b);
  UrukVector R=P;
  R.enqueue(o+1);
  UT_move(P,Q,1).apply(SCoP);
  UT_move(R,R,-1).apply(SCoP);
}
```

Figure 3.31: FISSION primitive

```
%transformation tile
%param BetaPrefix P
%param Integer k1
%param Integer k2
%prereq k1>0 && k2>0
%code
{
  Q=P.enclose();
  UT_stripmine(P,k2).apply(SCoP);
  UT_stripmine(Q,k1).apply(SCoP);
  UT_interchange(Q).apply(SCoP);
}
```

Figure 3.32: TILE primitive

only, i.e., without reference to the initial syntactic program. This allows for *one-time dependence analysis* before applying the transformation sequence, and *one-time check* at the very end, before code generation.

Let us briefly explain how this is achieved. Considering two distinct references to the same array in the program, at least one of them being a write, there is a dependence between them if their access functions coincide on some array element. Multiple refinement of this abstraction have been proposed, including dependence directions, distances, vectors and intervals [Wol96] to improve the precision about the localization of the actual dependences between run-time statement instances. In the polyhedral model, it is possible to refine this definition further and to compute an *exact* dependence information, as soon as all array references are affine [Fea91]. Exact dependences are classically captured by a system of affine inequalities over iteration vectors; when considering a syntactic loop nest, dependences at depth p between access functions F^S and F^T in statements S and T are exactly captured by the following union of polyhedra:

$$\mathcal{D}^S_{\mathrm{om}} \times \mathcal{D}^T_{\mathrm{om}} \cap \left\{ (\mathbf{i}^S, \mathbf{i}^T) \mid \mathrm{F}^S(\mathbf{i}^S) = \mathrm{F}^T(\mathbf{i}^T) \wedge \mathbf{i}^S \ll_p \mathbf{i}^T \right\},$$

where $\ll_p$ stands for the ordering of iteration vectors at depth p (i.e., equal component-wise up to depth $p-1$ and different at depth p).

Yet this characterization needs to be adapted to programs in a polyhedral representation, where no reference to a syntactic form is available, and where multiple schedule and domain transformations make the definition and tracking of the dependence information difficult. We thus replace the ordering on iteration vectors by the schedule-induced ordering, and split the constraints according to the decomposition of the schedule in our formalism. Two kinds of dependences at depth p can be characterized.

- Loop-carried dependence:

$$\beta^S_{0..p-1} = \beta^T_{0..p-1} \text{ and } (\mathrm{A}^S, \Gamma^S)\mathbf{i}^S \ll_p (\mathrm{A}^T, \Gamma^T)\mathbf{i}^T.$$

- Intra-loop dependence:

$$\beta^S[0..p-1] = \beta^T[0..p-1], \qquad ((\mathrm{A}^S, \Gamma^S)\mathbf{i}^S)_{0..p-1} = ((\mathrm{A}^T, \Gamma^T)\mathbf{i}^T)_{0..p-1} \text{ and } \beta^S_p < \beta^T_p.$$

Both kinds lead to a union of polyhedra that is systematically built, before any transformation is applied, for all pairs of references (to the same array) and for all depths (common to these references).

To solve the dependence tracking problem, we keep track of all modifications to the *structure* of the time and domain dimensions. In other words, we record any extension (dimension insertion, to implement, e.g., strip-mining) and any domain restriction (to implement, e.g., index-set splitting) into a work list, and we eventually traverse this list after all transformations have been applied to update dependence polyhedra accordingly. This scheme guarantees that the iteration domains and time dimensions correspond, after transformations, in the pre-computed dependence information and in the modified polyhedral program representation.

Dependence checking is implemented by intersecting every dependence polyhedron with the *reversed* schedule of the transformed representation. If any such intersection is non-empty, the resulting polyhedron captures the *exact set of dependence violations*. This step allows to derive the exact set of iteration vector pairs associated with causality constraints violations [VCBG06]. Based on this strong property, our implementation reports any dependence violation as a list of polyhedra; this report is very useful for automatic filtering of transformations in an iterative optimization framework, and as an optimization aid for the interactive user of URUK.

Interestingly, our formalism allows both dependence computation and checking to be simplified, relying on scalar comparisons on the β vectors to short-circuit complex polyhedral operations on inner depths. This optimization yields impressive speedups, due to the block-structured nature of most real-world schedules. The next section will explore such a real-world example and show a good scalability of this aggressive analysis.

3.4.4 URGenT: URUK Generation Tool

After polyhedral transformations, the (re)generation of imperative loop structures is the last step. It has a strong impact on the target code quality: we must ensure that no redundant guard or complex loop bound spoils performance gains achieved thanks to polyhedral transformations. We used the Chunky Loop Generator (CLooG), a recent Quilleré et al. method [QRW00] with some additional improvements to guarantee the absence of duplicated control [Bas04], to generate efficient control for full SPECfp2000 benchmarks and for SCoPs with more than 1700 statements. Polyhedral transformations make code generation particularly difficult because they create a large set of complex overlapping polyhedra that need to be scanned with do-loops [AI91, QRW00, Bas03, Bas04]. Because of the added complexity introduced, we had to design URGenT, a major reengineering of CLooG taking advantage of the normalization rules of our representation to bring exponential improvements to execution time and memory usage. The generated code size and quality greatly improved, making it better than typically hand-tuned code. [VBC06] details how URGenT succeeds in producing efficient code for a realistic optimization case-study in a few seconds only.

3.5 Semi-Automatic Optimization

Let us detail the application of our tools to the semi-automatic optimization of the swim benchmark, to show the effectiveness of the approach and the performance of the implementation on a representative benchmark. We target a 32bit and a 64bit architecture: an AMD Athlon XP 2800+ (Barton) at 2.08GHz with 512KB L2 cache and 512MB single-channel DDR SDRAM (running Mandriva Linux 10.1, kernel version 2.6.8), and a AMD Athlon 64 3400+ (ClawHammer) at 2.2GHz zith 1MB L2 cache and single-channel 1GB DDR SDRAM (running Debian GNU/Linux Sid, kernel version 2.6.11). The swim benchmark was chosen because it easily illustrates the benefits of implementing a sequence of transformations in our framework, compared to manual optimization of the program text, and because it presents a reasonably large SCoP to evaluate robustness (after fully inlining the three hot subroutines).

Figure 3.33 shows the transformation sequence for swim, implemented as a script for URUK. Syntactic compilation frameworks like PathScale EKOPath, Intel ICC and KAP implement a simplified form of this transformation sequence on swim, missing the fusion with the nested loops in subroutine `calc3`, which requires a very complex combination of loop peeling, code motion and three-level shifting. In addition, such a sequence is highly specific to swim and cannot be easily adapted, extended or reordered to handle other programs: due to syntactic restrictions of individual transformations, the sequence has to be considered as a whole since the effect of any of its components can hamper the application and profitability of the entire sequence. Conversely, within our semi-automatic framework, the sequence can be built without concern about the impact of a transformation on the applicability of subsequent ones. We demonstrate this through the dedicated transformation sequence in Figure 3.33.

This URUK script operates on the swim.N file, a persistent store of the compiler’s intermediate representation, dumped by EKOPath after interprocedural analysis and pre-optimization. At this step, EKOPath is directed to inline the three dominant functions of the benchmark, calc1, calc2 and calc3 (passing these function names to the -INLINE optimization switch). WRaP-IT processes the resulting file, extracting several SCoPs, the significant one being a section of 421 lines of code — 112 instructions in the polyhedral representation — in consecutive loop nests within the main function. Transformations in Figure 3.33 apply to this SCoP.

Labels of the form C*x*L*y* denote statement y of procedure calc*x*. Given a vector v and an integer $r \leq \dim v$, enclose(v,r) returns the prefix of length $\dim v - r$ of vector v (r is equal to 1 if absent). The primitives involved are the following: motion translates the β component (of a set of statements), shift translates the Γ matrix; peel splits the domain of a statement according to a given constraint and creates two labels with suffixes _1 and _2; stripmine and interchange are self-explanatory; and time-prefixed primitives mimic the effect of their iteration domain counterparts on time dimensions. Loop fusion is a special case of the motion primitive. Tiling is decomposed into double strip-mining and interchange. Loop unrolling (fullunroll) is delayed to the code generation phase.

Notice the script is quite concise, although the generated code is much more complex than the original swim benchmark (due to versioning, peeling, strip-mining and unrolling). In particular, loop fusion is straightforward, despite the fused loops domains differ by one or two iterations (due to peeling), and despite the additional multi-level shifting steps.

```
# Avoid spurious versioning
addContext(C1L1,’ITMAX>=9’)
addContext(C1L1,’doloop_ub>=ITMAX’)
addContext(C1L1,’doloop_ub<=ITMAX’)
addContext(C1L1,’N>=500’)
addContext(C1L1,’M>=500’)
addContext(C1L1,’MNMIN>=500’)
addContext(C1L1,’MNMIN<=M’)
addContext(C1L1,’MNMIN<=N’)
addContext(C1L1,’M<=N’)
addContext(C1L1,’M>=N’)

# Move and shift calc3 backwards
shift(enclose(C3L1),{’1’,’0’,’0’})
shift(enclose(C3L10),{’1’,’0’})
shift(enclose(C3L11),{’1’,’0’})
shift(C3L12,{’1’})
shift(C3L13,{’1’})
shift(C3L14,{’1’})
shift(C3L15,{’1’})
shift(C3L16,{’1’})
shift(C3L17,{’1’})
motion(enclose(C3L1),BLOOP)
motion(enclose(C3L10),BLOOP)
motion(enclose(C3L11),BLOOP)
motion(C3L12,BLOOP)
motion(C3L13,BLOOP)
motion(C3L14,BLOOP)
motion(C3L15,BLOOP)
motion(C3L16,BLOOP)
motion(C3L17,BLOOP)
```

```
# Peel and shift to enable fusion
peel(enclose(C3L1,2),’3’)
peel(enclose(C3L1_2,2),’N-3’)
peel(enclose(C3L1_2_1,1),’3’)
peel(enclose(C3L1_2_1_2,1),’M-3’)
peel(enclose(C1L1,2),’2’)
peel(enclose(C1L1_2,2),’N-2’)
peel(enclose(C1L1_2_1,1),’2’)
peel(enclose(C1L1_2_1_2,1),’M-2’)
peel(enclose(C2L1,2),’1’)
peel(enclose(C2L1_2,2),’N-1’)
peel(enclose(C2L1_2_1,1),’3’)
peel(enclose(C2L1_2_1_2,1),’M-3’)
shift(enclose(C1L1_2_1_2_1),{’0’,’1’,’1’})
shift(enclose(C2L1_2_1_2_1),{’0’,’2’,’2’})

# Double fusion of the three nests
motion(enclose(C2L1_2_1_2_1),TARGET_2_1_2_1)
motion(enclose(C1L1_2_1_2_1),C2L1_2_1_2_1)
motion(enclose(C3L1_2_1_2_1),C1L1_2_1_2_1)

# Register blocking and unrolling (factor 2)
stripmine(enclose(C3L1_2_1_2_1,2),2,2)
stripmine(enclose(C3L1_2_1_2_1,1),4,2)
interchange(enclose(C3L1_2_1_2_1,2))
fullunroll(enclose(C3L1_2_1_2_1,2))
fullunroll(enclose(C3L1_2_1_2_1,1))
```

Figure 3.33: URUK script to optimize swim

The application of this script is fully automatic; it produces a significantly larger code of 2267 lines, roughly one third of them being naive scalar copies to map schedule iterators to domain ones, fully eliminated by copypropagation in the subsequent run of EKOPath or Open64. This is not surprising since most transformations in the script require domain decomposition, either explicitly (peeling) or implicitly (shifting prolog/epilog, at code generation). It takes 39s to apply the whole transformation sequence up to native code generation on a 2.08GHz AthlonXP. Transformation time is dominated by back-end compilation (22s). Polyhedral code generation takes only 4s. Exact polyhedral dependence analysis (computation and checking) is acceptable (12s). Applying the transformation sequence itself is negligible. These execution times are very encouraging, given the complex overlap of peeled polyhedra in the code generation phase, and since the full dependence graph captures the exact dependence information for the 215 array references in the SCoP at every loop depth (maximum 5 after tiling), yielding a total of 441 dependence matrices. The result of this application is a new intermediate representation file, sent to EKOPath or Open64 for further scalar optimizations and native code generation.

Compared to the *peak performance attainable by the best available compiler*, PathScale EKOPath (V2.1) with the best optimization flags,[6] our tool achieves **32% speedup on Athlon XP and 38% speedup on Athlon 64**. Compared to the *base SPEC* performance numbers,[7] our optimization achieves **51% speedup on Athlon XP and 92% speedup on Athlon 64**. We are not aware of any other optimization effort — manual or automatic — that brought swim to this level of performance on x86 processors.[8]

We do not have results on IA64 yet, due to back-end instability issues in Open64 (with large basic blocks). We expect an additional level of tiling and more aggressive unrolling will be profitable (due to the improved TLB management, better load/store bandwitdh and larger register file on Itanium 2 processors).

Additional transformations need to be implemented in URUK to authorize semi-automatic optimization of a larger range of benchmarks. In addition, further work on the iterative optimization driver is being conducted to make this process more automatic and avoid the manual implementation of an URUK script. Yet the tool in its current state is of great use for the optimization expert who wishes to quickly evaluate complex sequences of transformations.

The following section proposes to further automate the process of building such a sequence of loop transformations, reducing the complexity of the optimization problem for the expert programmer or for a fully automatic optimization heuristic.

3.6 Automatic Correction of Loop Transformations

Program optimization is a combinatorial problem, most decision sub-problems being undecidable and their simplified, statically tractable models, NP-hard. With feedback-directed techniques, every program — and sometimes every program with every different input — requires a different tuning of optimization parameters and ordering. The complexity of the optimization search space results from the intrinsic complexity of the target architecture, and from the characterization of legal program transformations. Our work allows to simplify and narrow this monstrous optimization search space, addressing the latter source of complexity (the characterization of legal transformations), and focusing on regular nested loops in imperative programs.

The power of an automatic optimizer or parallelizer greatly depends on its capacity to decide whether two portions of the program execution may be interchanged or run in parallel. Such knowledge is related to the difficult task of *dependence analysis* which aims at precisely disambiguating memory references. Without special care, searching for loop transformations amounts to traversing a many-dimensional vector space with a Monte Carlo method, using dependence conditions as a filter. This approach may be applicable to the selection of two or three classical loop transformations, but does not scale to real problems, where sequences of tens or hundreds transformations are common [CGP$^+$05].

Within the polyhedral model, and relying on profound results from the duality theory [Sch86], it is possible to directly characterize all affine schedules associated with the legal transformations of a loop nest, as a finite union of polyhedra. The main algorithm to look for a "good" schedule in this set has been proposed by Feautrier [Fea92]. This algorithm — and its main extensions and improvements [LL97, LLL01, TVSA01, BF04] — relies on simplistic linear optimization models and suffers from multiple sources of combinatorial complexity:

1. the number of polyhedra it considers is exponential with respect to the program size;

[6] Athlon XP: `-m32 -Ofast -OPT:ro=2:Olimit=0:div_split=on:alias=typed -LNO:fusion=2:prefetch=2 -fno-math-errno`; Athlon 64 (in 64 bits mode): `-march=athlon64 -LNO:fusion=2:prefetch=2 -m64 -Ofast -msse2 -lmpath`; `pathf90` always outperformed Intel ICC by a small percentage.

[7] With optimization flag `-Ofast`.

[8] Notice we consider the SPEC 2000 version of swim, much harder to optimize through loop fusion than the SPEC 95 version.

2. the dimensionality of polyhedra is proportional to the program size, which incurs an exponential complexity in the integer linear programs the algorithm relies on.

Radical improvements are needed to scale these algorithms to real-size loop nests (a few hundred statements or more), and to complement linear programming with more pragmatic empirical operation research heuristics. We are working along these lines [PBCV07], but it is unreasonable to expect this approach will scale alone to full-size programs.

We thus develop a complementary approach to reduce the complexity and size of the transformation space. It consists in narrowing the set of variables of this combinatorial problem. Intuitively, operation research heuristics are focused on the "hard to find" *and* "most performance impacting" ones, relying on a simple model to optimize across other dimensions of the search space.

Our main technical contribution is an algorithm to exploit *dependence violations*: by identifying exactly the violated dependences, we derive *an automatic correction scheme to fix an illegal transformation sequence* with "minimal changes". This correction scheme amounts to decomposing the iteration domain and translating (a.k.a. shifting) the schedule on each part (by a minimal amount) so that dependences are satisfied. Our algorithm has two important qualities:

1. it is sound and *complete*: any multidimensional affine schedule will be corrected as long as dependence violations can be corrected by translation;

2. it applies a *polynomial* number of operations on dependence polyhedra (each one may be exponential, but does not depend on the program size).

We will demonstrate the effectiveness and scalability of our automatic correction scheme through the optimization of two of these benchmarks: `swim` and `mgrid`.

3.6.1 Related Work and Applications

The concept of *enabling* transformation is central to design of loop optimizers [AK02]. Very often, a locality or parallelism enhancing transformation violates the data-flow semantics of the program, although a simple preconditioning step on the operational semantics — typically the program schedule or its storage management — suffices to enable it. Loop shifting or pipelining [DH00], and loop skewing [Ban92] are such enabling transformations of the schedule, while loop peeling isolates violations on boundary conditions [Muc97, FGL99] and variable renaming or privatization removes dependences [CFR$^+$91, MAL93, KS98]. Unfortunately, except in special cases (e.g., the unimodular transformation framework, or acyclic dependence graphs), one faces the decision problem of choosing an enabling transformation with no guarantee that it will do any good.

The induced combinatorics of these decision problems drive the search for a more compositional approach, where transformation legality is guaranteed by construction, using the Farkas lemma on affine schedules. As stated in the introduction, this is our main motivation, but taking all legality constraints into account makes the algorithms not scalable, and suggest staging the selection of a loop nest transformation into an "approximately correct" combinatorial step — addressing the main performance concerns — and a second *correction* step. The idea of applying "after-thought" corrections on affine schedules was first proposed by Bastoul [BF05]. However, it still relies on a projection of the schedule on the subspace of legal Farkas coefficients and does not scale with program size.

We follow the idea of correcting illegal schedules, but focus on a particular kind of correction that generalizes loop *shifting*, *fusion* and *distribution* [AK02]. These three classical (operational) program transformations boil down to the same algebraic translation in multidimensional affine scheduling. These are some of the most important enabling transformations in the context of automatic loop parallelization [AK02, DH00]. They induce less code complexity than, e.g., loop skewing (which may degrade performance due to the complexity of loop bound expressions), and can cope with a variety of cyclic dependence graphs. Yet the problem, seen as a decision one, has been proved NP-complete by Darte [DH00]. This is due to the inability to characterize data-parallelism in loops in a linear fashion (many data-parallel programs cannot be expressed with multi-dimensional schedules [LL97]). Although we use the same tools as Darte to correct schedules, we avoid this pitfall stating our problem as a linear optimization (using the Bellman-Ford algorithm [CLR89]).

Decoupling research on the linear and constant parts of the schedule has been proposed earlier [DSV97, DH00, VBJC03]. These techniques all use simplified representation of dependences (i.e., dependence vectors) and rely on finding enabling transformations for their specific purpose, driven by optimization heuristics. Our approach is more general as it applies to any potentially illegal affine schedule.

Alternatively, Crop and Wilde [CW99] and Feautrier [Fea06] attempt to reduce the complexity of affine scheduling with a modular or structural decomposition. These algorithms are very effective, but still resort to solving large, non-scalable integer-linear programs. They are complementary to our correction scheme.

To sum-up, our approach has two main complexity advantages.

1. The number of dimensions of the optimization search space is reduced. Depending on the optimization strategy and the program, it may eliminate from one third to almost all of the unknowns in the optimization problem. We also show that remaining dimensions have a lower contribution to the size of the search space and a higher impact on performance than the ones that can be eliminated.

2. Index-set splitting is the state-of-the-art technique to decompose iteration domains, allowing more expressive piecewise affine schedules to be built. Its current formulation is a complex decision problem [FGL99] which requires interaction with a non scalable scheduling algorithm. Our approach replaces it by a simple heuristic, combined with the automatic correction procedure. This is extremely helpful, considering that each decision problem has a combinatorial impact on the phase ordering, selection and parameterization problem associated with the feedback-directed optimization of each application.

Finally, our results also facilitate the design of domain-specific program generators [WPD00, FJ98, PSX$^+$04, DBR$^+$05, GVB$^+$06, CDG$^+$06], a very pragmatic solution to the design of portable optimized libraries. Indeed, although human-written code is concise (code factoring in loops and functions), optimizing it for modern architectures incurs several code size expansion steps, including function inlining, specialization, loop versioning and unrolling. Domain-specific program generators (also known as active libraries [LBCO03]) rely on feedback-directed optimization and iterative search to generate nearly optimal library or application code. Our approach reduces the number of transformation steps, focusing the optimization problem to the core execution anomalies. Since most steps are enabling transformations that enable key optimizations [PTCV04], it is quite beneficial to the productivity and comfort of designers of such program generators.

3.6.2 Dependence Analysis

Array dependence analysis is the starting point for any polyhedral optimization. It computes non transitively-redundant, iteration vector to iteration vector, directed dependences [Fea88a, VCBG06]. In order to correct dependence violations, it is first needed to *compute the exact dependence information between every pair of instances*, i.e., every pair of statement iterations. Considering a pair of statements S and T accessing memory locations where at least one of them is a write, there is a dependence from an instance $\langle S, \mathbf{i}^S\rangle$ of S to an instance $\langle T, \mathbf{i}^T\rangle$ of T (or $\langle T, \mathbf{i}^T\rangle$ depends on $\langle S, \mathbf{i}^S\rangle$) if and only if the following *instancewise* conditions are met:

Execution condition: both instances belong to the corresponding statement iteration domain: $D^S_{\mathbf{i}} \cdot \mathbf{i}^S \geq 0$ and $D^T_{\mathbf{i}} \cdot \mathbf{i}^T \geq 0$,

Conflict condition: both instances refer the same memory location: $\left(\mathrm{Acc}^S_{\mathbf{i}} \mid \mathrm{Acc}^S_{\mathbf{i}_{\mathrm{gp}}}\right) \cdot \mathbf{i}^S = \left(\mathrm{Acc}^T_{\mathbf{i}} \mid \mathrm{Acc}^T_{\mathbf{i}_{\mathrm{gp}}}\right) \cdot \mathbf{i}^T$, and

Causality condition: the instance $\langle S, \mathbf{i}^S\rangle$ is executed before $\langle T, \mathbf{i}^T\rangle$ in the original execution: $\Theta^S \cdot \mathbf{i}^S \ll \Theta^T \cdot \mathbf{i}^T$,

where $\ll$ denotes the lexicographic order on vectors.

The schedule (the multidimensional time-stamp) at which an instance is executed is determined, for statement S, by the $2d_S + 1$ vector given by $\Theta^S \cdot \mathbf{i}^S$. Relative order between instances is given by the relative lexicographic order of their schedule vectors.

Consider the original polyhedral representation of a program, before any transformation has been applied. For a given statement S, matrix A^S is the identity, Γ^S is 0 and vector β^S captures the syntactic position of S in the original code. In this configuration, the three aforementioned conditions correspond to the classical definition of polyhedral dependences [Fea88a, Pug91b].

A dependence is said to be loop independent of depth $p \leq 0$ if the causality condition $\Theta^S \cdot \mathbf{i}^S \ll \Theta^T \cdot \mathbf{i}^T$ is resolved sequentially on the $\beta_{|p|}$ component of the schedule. A dependence is said to be loop carried at loop depth $p > 0$ if the causality condition is resolved by the $(\mathrm{A} \mid \Gamma)$ component of the schedule at depth p.

Loop-carried dependence at depth $p > 0$:

$$\left((\mathrm{A}^S \mid \Gamma^S) \cdot \mathbf{i}^S\right)_{0..p-1} = \left((\mathrm{A}^T \mid \Gamma^T) \cdot \mathbf{i}^T\right)_{0..p-1},$$
$$\beta^S_{0..p-1} = \beta^T_{0..p-1} \text{ and } \left((\mathrm{A}^S \mid \Gamma^S) \cdot \mathbf{i}^S\right)_p < \left((\mathrm{A}^T \mid \Gamma^T) \cdot \mathbf{i}^T\right)_p$$

Loop-independent dependence at depth $p \leq 0$:

$$\begin{aligned}&\left((\mathrm{A}^S \mid \Gamma^S)\cdot \mathbf{i}^S\right)_{0..|p|-1} = \left((\mathrm{A}^T \mid \Gamma^T)\cdot \mathbf{i}^T\right)_{0..|p|-1},\\ &\beta^S_{0..|p|-1} = \beta^T_{0..|p|-1} \text{ and } \beta^S_{|p|} < \beta^T_{|p|}.\end{aligned}$$

The purpose of dependence analysis is to compute a directed dependence multi-graph DG. Unlike traditional reduced dependence graphs, an arc $S \to T$ in DG is labeled by a polyhedron capturing the set of iteration vector pairs ($\mathbf{i}^S$, $\mathbf{i}^T$) in dependence. These pairs belong to the Cartesian product space $\mathrm{P}^{\mathrm{S,T}}$ of dimension $(d_S + d_{\mathrm{gp}} + 1) + (d_T + d_{\mathrm{gp}} + 1)$ and meet the instancewise dependence conditions. Since the global parameters are invariant across the whole SCoP, we can remove redundant parameter dimensions and project this space into the equally expressive one of dimension $d_S + d_T + d_{\mathrm{gp}} + 1$.

3.6.3 Characterization of Violated Dependences

After transforming the SCoP, the question arises whether the resulting program still executes correct code. Our approach consists in saving the dependence graph, before applying any transformation, then to apply a given transformation sequence, and eventually to run a legality analysis at the very end of the sequence.

We consider a dependence from S to T in the original code and we want to determine if it has been preserved in the transformed program.

The *violated dependence analysis* [VCBG06] efficiently computes the iterations of the Cartesian product space $\mathrm{P}^{\mathrm{S,T}}$ that were in a dependence relation in the original program and *whose order has been reversed by the transformation*. These iterations, should they exist, do not preserve the causality of the original program. Let $\delta^{S\to T}$ denote the dependence polyhedron from S to T; we are looking for the exact set of iterations of $\delta^{S\to T}$ such that there is a dependence from T to S at transformed depth p. By reasoning in the transformed space, it is straightforward to see that the set of iterations that violate the causality condition is the intersection of a dependence polyhedron with the constraint set $\Theta^S \cdot \mathbf{i}^S \geq \Theta^T \cdot \mathbf{i}^T$. This gives rise to the case distinction of Figure 3.34 if $p > 0$, and of Figure 3.35 if $p \leq 0$[9]. Note that for the case $p \leq 0$, the violated dependence is actually the set of iterations that are *potentially* in violation (i.e.,that have the same timestamp up to depth p). The additional constraint $\beta^S_{|p|} > \beta^T_{|p|}$ is also needed.

$$\left(\begin{array}{c|c|c} \multicolumn{3}{c}{\delta^{S\to T}} \\ \hline -\mathrm{A}^S_{1..p-1,\bullet} & \mathrm{A}^T_{1..p-1,\bullet} & -\Gamma^S_{1..p-1,\bullet} + \Gamma^T_{1..p-1,\bullet} = 0 \\ \hline \mathrm{A}^S_{p,\bullet} & -\mathrm{A}^T_{p,\bullet} & \Gamma^S_{p,\bullet} - \Gamma^T_{p,\bullet} \geq 1 \end{array}\right)$$

Figure 3.34: Violated dependence at depth $p > 0$

$$\left(\begin{array}{c|c|c} \multicolumn{3}{c}{\delta^{S\to T}} \\ \hline \mathrm{A}^S_{1..p,\bullet} & -\mathrm{A}^T_{1..p,\bullet} & \Gamma^S_{1..p,\bullet} - \Gamma^T_{1..p,\bullet} = 0 \end{array}\right)$$

Figure 3.35: Violated dependence candidates at depth $p \leq 0$

A violated dependence polyhedron at depth p, as defined in Figure 3.34, will be referred to as $\delta_{\mathrm{v}}{}_p^{S\to T}$. The prerequisites on $\beta_{1..|p|-1}$ are the same as for the dependence analysis outlined in Section 3.6.2 since we are essentially solving the *same* problem in the transformed space.

We also define a slackness polyhedron at depth p which contains the set of points originally in dependence and that are still executed in correct order after transformation. Such a polyhedron will be referred to as $\delta_{\mathrm{s}}{}_p^{S\to T}$ and is built like $\delta_{\mathrm{v}}{}_p^{S\to T}$ with the sole exception of the last row of figure. 3.34 which is replaced by:

$$\mathrm{A}^S_{p,\bullet} - \mathrm{A}^T_{p,\bullet} + \Gamma^S_{p,\bullet} - \Gamma^T_{p,\bullet} \leq 0$$

Finally, to avoid enforcing unnecessary constraints in reductions or scans [AK02], it is also possible to consider fundamental properties such as commutativity and associativity, hence further refine the violated dependence graph.

[9] We write $\mathrm{A}_{p,\bullet}$ to express the p^{th} line of A and $\mathrm{A}_{1..p-1,\bullet}$ for lines 1 to $p-1$

3.6.4 Correction by Shifting

We propose a greedy algorithm to incrementally correct violated dependences, from the outermost to the innermost nesting depth, with a combination of fusion and shifting. Fusions will be performed to correct loop-independent violations ($p \leq 0$) while shifts will target loop-carried violations ($p > 0$).

Violation and Slackness

At depth p, the question raises whether a subset of those iterations — whose dependence has not yet been fully resolved up to current depth — are strictly in violation and must be corrected.

When correcting loop-carried violations, we define the following affine functions from $\mathbb{Z}^{d_S+d_T+d_{gp}+1}$ to $\mathbb{Z}^{d_{gp}+1}$:

- $\Delta_{\mathrm{v}}\Theta_p^{S\to T} = \mathrm{A}^S_{p,\bullet} - \mathrm{A}^T_{p,\bullet} + \Gamma^S_{p,\bullet} - \Gamma^T_{p,\bullet}$ which computes the amount of time units at depth p by which a specific instance of T executes before one of S.
- $\Delta_{\mathrm{s}}\Theta_p^{S\to T} = -\mathrm{A}^S_{p,\bullet} + \mathrm{A}^T_{p,\bullet} - \Gamma^S_{p,\bullet} + \Gamma^T_{p,\bullet}$ which computes the amount of time units at depth p by which a specific instance of S executes before one of T.

Then, let us define the parametric extremal values of these two functions:

- $Shift^{S\to T} = \max_{\left(X\in \mathrm{P}^{\mathrm{S,T}}\right)} \left\{\Delta_{\mathrm{v}}\Theta_p^{S\to T}\cdot X > 0 \mid X \in \delta_{\mathrm{v}}{}_p^{S\to T}\right\}$
- $Slack^{S\to T} = \min_{\left(X\in \mathrm{P}^{\mathrm{S,T}}\right)} \left\{\Delta_{\mathrm{s}}\Theta_p^{S\to T}\cdot X \geq 0 \mid X \in \delta_{\mathrm{s}}{}_p^{S\to T}\right\}$

We use the parametric integer linear program solver PIP [Fea88b] to perform these computations. The result is a piecewise, quasi-affine (affine with additional parameters to encode modulo operations) function of the parameters. It is characterized by a disjoint union of polyhedra where this function is quasi-affine. This piecewise affine function is encoded as a *parametric quasi-affine selection tree* — or *quast* — [Fea88b, Fea88a].

The loop-independent case is much simpler. Each violated dependence must satisfy conditions on β:

- if $\delta_{\mathrm{v}}{}_p^{S\to T}$ not empty and $\beta^S_{|p|} > \beta^T_{|p|}$ $Shift^{S\to T} = \beta^S_{|p|} - \beta^T_{|p|}$
- if $\delta_{\mathrm{v}}{}_p^{S\to T}$ not empty and $\beta^S_{|p|} \leq \beta^T_{|p|}$ $Slack^{S\to T} = \beta^S_{|p|} - \beta^T_{|p|}$

The correction problem can then be reformulated as finding a solution to a system of differential constraints on parametric quasts. For any statement S, we shall denote by c_S the unknown amount of correction: a piecewise, quasi-affine function; c_S is called the *shift amount* for statement S. The problem to solve becomes:

$$\forall (S,T) \in SCoP \qquad \left\{ \begin{array}{c} c_T - c_S \leq -Shift^{S\to T} \\ c_T - c_S \leq Slack^{S\to T} \end{array} \right\}$$

Such a problem can be solved with a variant of the Bellman-Ford algorithm [CLR89].

With piecewise quasi-affine functions (quasts) labeling the edges of the graph, the question of parametric case distinction arises when considering addition and maximization of the shift amounts resulting from different dependence polyhedra. Also for correctness proofs of the Bellman-Ford algorithm to hold, triangular inequalities and transitivity of the $\leq$ operator on quasts must also hold. The algorithm in Figure 3.36 allows to maintain case disjunction while enforcing all the required properties for any configuration of the parameters.

At each step of the separation algorithm, two cases are computed and tested for emptiness. Step 12 checks if under some configuration of the parameters given by $Shift_1^{cond} \wedge Shift_2^{cond}$, the quantity $Shift_1$ is the biggest. Step 15 implements the complementary check.[10]

As an optimization, it is often possible to extend disjoint conditionals by continuity, which reduces the number of versions (associated with different parameter configurations), hence reduce complexity and the resulting code size. For example:

$$\left\{ \begin{array}{c} \texttt{if } (M=3) \texttt{ then } 2 \\ \texttt{else if } (M\geq 4) \texttt{ then } M-1 \end{array} \right\} \equiv \texttt{if } (M\geq 3) \texttt{ then } M-1.$$

Experimentally, performing this post processing allows up to 20% less versioning at each correction depth. Given the multiplicative nature of duplications, this can translate into exponentially smaller generated code.

[10] Unlike the more costly separation algorithm by Quilleré [QR99] used for code generation, this one only needs intersections (no complementation or difference).

```
SeparateMinShifts: Eliminate redundancy in a list of shifts
Input:
  redundantlist: list of redundant shift amounts and conditions
Output: non redundant list of shift amounts and conditions
  resultinglist ← empty list
  while(redundantlist not empty)
    if(resultinglist is empty)
      resultinglist.append(redundantlist.head)
      redundantlist ← redundantlist.tail
    else
      tmplist ← empty list
      $Shift_1$ ← redundantlist.head
      redundantlist ← redundantlist.tail
      while(resultinglist not empty)
        $Shift_2$ ← resultinglist.head
        resultinglist ← resultinglist.tail
        cond1 ← $Shift_1^{cond} \wedge Shift_2^{cond} \wedge (Shift_2 < Shift_1)$
        if(cond1 not empty)
          tmplist.append(cond1, $Shift_1$)
        cond2 ← $Shift_1^{cond} \wedge Shift_2^{cond} \wedge (Shift_2 \geq Shift_1)$
        if(cond2 not empty)
          tmplist.append(cond2, $Shift_2$)
        if(cond1 is empty and cond2 is empty)
          tmplist.append($Shift_1^{cond}$, $Shift_1$)
          tmplist.append($Shift_2^{cond}$, $Shift_2$)
       resultinglist ← tmplist
 return resultinglist
```

Figure 3.36: Conditional separation

Constraints graphs

In this section we will quickly outline the construction of the constraints graph used in our correction algorithm. For a given depth p, the violated dependence denoted by VDG$_p$ is a directed multigraph where each node represents a statement in the SCoP.

The construction of the violated dependence graph proceeds as follows. For each violated polyhedron $\delta\mathrm{v}_p^{S\rightarrow T}$, the minimal necessary correction is computed and results in a parametric conditional and a shift amount. We add an edge, between S and T in VDG_p of type $\mathcal{V}$ (violation). Such an edge is decorated with the tuple ($\delta\mathrm{v}_p^{S\rightarrow T}$, $Shift^{Cond}$, -$Shift^{S\rightarrow T}$).

When $\mathcal{V}$ type arcs are considered, they bear the minimal shifting requirement by which T *must* be shifted for the corrected values of Θ^S and Θ^T to nullify the violated polyhedron $\delta\mathrm{v}_p^{S\rightarrow T}$. Notice however that shifting T by $Shift^{S\rightarrow T}$ amount does not fully solve the dependence problem. It solves it for the subset of points $\{X \in \delta\mathrm{v}_p^{S\rightarrow T} \mid \Delta_\mathrm{v}\Theta_p^{S\rightarrow T} < Shift^{S\rightarrow T}\}$. The remaining points — the facet of the polyhedron such that $\{X \in \delta\mathrm{v}_p^{S\rightarrow T} \mid \Delta_\mathrm{v}\Theta_p^{S\rightarrow T} = Shift^{S\rightarrow T}\}$ — are carried for correction at the next depth and will eventually be solved at the innermost β level, as will be seen shortly.

For VDG$_p$ such that $p \leq 0$, the correction is simply done by reordering the $\beta_{|p|}$ values. No special computation is necessary as only the relative values of $\beta_{|p|}^S$ and $\beta_{|p|}^T$ are needed to determine the sequential order. When such a violated, it means the candidate dependence polyhedron $\delta\mathrm{v}_p^{S\rightarrow T}$ is not empty *and* $\beta_{|p|}^S > \beta_{|p|}^T$. The correction algorithm forces the synchronization of the loop independent schedules by setting $\beta_{|p|}^S = \beta_{|p|}^T$ and carries $\delta\mathrm{v}_p^{S\rightarrow T}$ to be corrected at depth $|p|+1 > 0$.

Of course, for the special case where $|p| = \dim(\beta_p^S)$ or $|p| = \dim(\beta_p^T)$, there is *no next* depth available, hence the solution of strictly ordering $\beta_{|p|}$ values is chosen: remaining violated dependences *have to* be corrected.

For each original dependence $\delta^{S\rightarrow T}$ in the DG that has not been completely solved up to current depth (i.e.,the candidate violated polyhedron constructed in figure. 3.35 is not empty and $\delta v_p^{S\to T}$ is empty); add an edge, between S and T in VDG_p of type $\mathcal{S}$ (slackness). Such an edge is decorated with the tuple ($\delta v_p^{S\to T}$, $Slack^{Cond}$, $Slack^{S\to T}$).

When $\mathcal{S}$ type edges are considered, they bear the maximal shifting allowed for S so that the causality relation $S \to T$ is ensured. If at any time a node is shifted by a quantity bigger than one of the maximal allowed outgoing slacks, it will give rise to new outgoing shift edges.

We are now ready to describe the greedy correction algorithm.

The Algorithm

The correction algorithm is interleaved with the incremental computation of the VDG at each depth level. The fundamental reason is that corrections at previous depths need to be taken into account when computing the violated dependence polyhedra at the current level.

```
CorrectLoopDependentNode: Corrects a node by shifting
Input:
  node: A node in $VDG_p$
Output: A list of parametric shift amounts and conditionals
for the node
  corrections ← corrections of node already
                           computed in previous passes
foreach(edge (S, node, Vi) incoming into node)
  compute minimal $Shift^{S\to node}$ and $Shift^{Cond}$
  corrections.append($Shift^{S\to node}$, $Shift^{Cond}$)
if(corrections.size > 1)
  corrections ← SeparateMinShifts(corrections)
foreach(edge (node, T) outgoing from node)
  foreach(corr in corrections)
    compute a new shift $\delta v_p^{node\to T}$ using corr for node
    if($\delta v_p^{node\to T}$ not empty)
      addedge(node, T, $\mathcal{V}$, $\delta v_p^{node\to T}$) to $VDG_p$
    else
      compute a new slack $\delta s_p^{node\to T}$ using corr for node
      addedge(node, T, , $\mathcal{S}$, $\delta s_p^{node\to T}$) to $VDG_p$
   removeedge(edge) from $VDG_p$
return corrections
```

Figure 3.37: Shifting a node for correction

```
CorrectLoopDependent: Corrects a VDG by shifting
Input:
  VDG: A node in $VDG_p$
Output: A list of parametric shift amounts and conditionals
for the node
  corrections ← empty list
for(i rightarrow to $|V| - 1$))
  nodelist → nodes(VDG) with incoming edge of type $\mathcal{V}$
  foreach(node in nodelist)
    corrections.append(CorrectLoopDependentNode(node))
 return corrections
```

Figure 3.38: Correcting a VDG

```
**CorrectSchedules**: Corrects an illegal schedule
*Input*:
  program: A program in URUK form
  dependences: The list of polyhedral dependences of the program
*Output*: Corrected program in URUK form
Build $VDG_0$
correctionList ← CorrectLoopIndependent($VDG_0$)
commit correctionList
**for**(p=1; p<=$\max_{(S\ \ in\ \ SCoP)}\{rank(\beta_S)\}$)
  Build $VDG_p$
  correctionList ← CorrectLoopDependent($VDG_p$)
  commit correctionList
  Build $VDG_{-p}$
  correctionList ← CorrectLoopIndependent($VDG_{-p}$)
commit correctionList
```

Figure 3.39: Correction algorithm

The main idea for depths $p > 0$ is to shift targets of violated dependences by the *minimal shifting amount necessary*. If any of those incoming shifts is bigger than any outgoing slack. The outgoing slacks turn into new violations that need to be corrected. During the graph traversal, any node may be traversed at most $|V| - 1$ times. At each traversal, we gather the previously computed corrections along with incoming violations and we apply the separation phase of Figure 3.36.

For loop-carried dependences, the algorithm to correct a node is outlined in Figure 3.37.

We use an incrementally growing cache to speed up polyhedral computations, as proposed in [VCBG06]. Step 8 of Figure 3.37 uses PIP to compute the minimal incoming shift amount; it may introduce case distinctions, and since we label edges in the VDG with single polyhedra and not quasts, it may result in inserting new outgoing edges. When many incoming edges generate many outgoing edges, Step 11 separates these possibly redundant amounts using the algorithm formerly given in Figure 3.36. In practice, PIP can also introduce new modulo parameters for a certain class of ill-behaved schedules. These must be treated with special care as they will expand d_{gp} and are generally a hint that the transformation will eventually generate code with many internal modulo conditionals.

For loop-independent corrections on the other hand, all quantities are just integer differences, without any case distinction. The much simpler algorithm is just a special case.

The full correction algorithm is given in Figure 3.39. Termination and soundness of this algorithm are straightforward, from the termination and soundness of the Bellman-Ford version [CLR89] applied successively at each depth.

Lemma 5 (Depth-*p* Completeness) *If there exist correction shift amounts satisfying the system of violation and slackness constraints at depth p, then the correction algorithm removes all violations at depth p.*

The proof derives from the completeness of Bellman-Ford's algorithm. Determining the minimal correcting shift amounts to computing the maximal value of linear multivariate functions ($\Delta_v\Theta_p^{S\to T}$ and $\Delta_s\Theta_p^{S\to T}$) over bounded parameterized convex polyhedra (${\delta_v}_p^{S\to T}$ and ${\delta_s}_p^{S\to T}$). This problem is solved in the realm of parametric integer linear programming. The separation algorithm ensures equality or incompatibility of the conditionals enclosing the different amounts. The resulting quasts therefore satisfy the transitivity of the operations of max, min, $+$ and $\leq$. When VDG_p has no negative weight cycle, the correction at depth p succeeds; the proof is the same as for the Bellman-Ford algorithm and can be found in [CLR89].□

As mentioned earlier, the computed shift amounts are the minimal necessary so that the schedule at a given depth is not violated after correction. The whole dependence at each step is not fully resolved and is carried for correction at the next level.

Another solution would be to shift target statements by $Shift^{S\to T} + 1$, but this is deemed too intrusive. Indeed, this amounts to adding -1 to all negative edges on the graph, potentially making the correction impossible.

The question arises whether a shift amount chosen at a given depth may interfere with the correction algorithm at a higher depth. The following property guarantees this is not the case.

```
    for (i=0; i<=N; i++)
S1  | A[i] = ...;
S2  A[1] = A[5];
    for (i=0; i<=N; i++)
S3  | B[i] = A[i+1];
```

$A_{S_1} = [1]$	$A_{S_2} = [1]$	$A_{S_3} = [1]$
$\beta_{S_1} = [0,0]$	$\beta_{S_2} = [1,0]$	$\beta_{S_3} = [2,0]$
$\Gamma_{S_1} = [0,0]$	$\Gamma_{S_2} = [0,0]$	$\Gamma_{S_3} = [0,0]$

Figure 3.40: Original code

```
    for (i=0; i<=N; i++)
S3  | B[i] = A[i+1];
S2  A[1] = A[5];
    for (i=0; i<=N; i++)
S1  | A[i] = ...;
```

$A_{S_1} = [1]$	$A_{S_2} = [1]$	$A_{S_3} = [1]$
$\beta_{S_1} = [2,0]$	$\beta_{S_2} = [1,0]$	$\beta_{S_3} = [0,0]$
$\Gamma_{S_1} = [0,0]$	$\Gamma_{S_2} = [0,0]$	$\Gamma_{S_3} = [0,0]$

Figure 3.41: Illegal schedule

```
    for (i=0; i<=N; i++)
S2  | if (i==0)  A[1] = A[5];
S3  | B[i] = A[i+1];
S1  | A[i] = ...;
```

$A_{S_1} = [1]$	$A_{S_2} = [1]$	$A_{S_3} = [1]$
$\beta_{S_1} = [2,0]$	$\beta_{S_2} = [2,0]$	$\beta_{S_3} = [2,0]$
$\Gamma_{S_1} = [0,0]$	$\Gamma_{S_2} = [0,0]$	$\Gamma_{S_3} = [0,0]$

Figure 3.42: After correcting $p = 0$

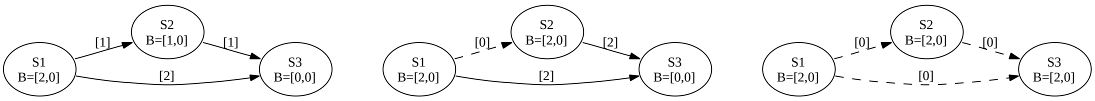


Figure 3.43: Outline of the correction for $p = 0$

```
    for (i=0; i<=N; i++)
S1  | A[i] = ...;
S21 | if (i==1 && N<=4)  A[1] = A[5];
S22 | if(i==5 && N>=5)  A[1] = A[5];
S3  | B[i] = A[i+1];
```

$A_{S_1} = [1]$	$A_{S_{21}} = [1]$	$A_{S_{22}} = [1]$	$A_{S_3} = [1]$
$\beta_{S_1} = [2,0]$	$\beta_{S_{21}} = [2,0]$	$\beta_{S_{22}} = [2,0]$	$\beta_{S_3} = [2,0]$
$\Gamma_{S_1} = [0,0]$	$\Gamma_{S_{21}} = [0,0]$	$\Gamma_{S_{22}} = [0,5]$	$\Gamma_{S_3} = [0,0]$

Figure 3.44: Correcting + versioning S_2

```
    for (i=0; i<=N; i++)
S1  | A[i] = ...;
S21 | if (i==1 && N<=4)  A[1] = A[5];
S22 | if (i==5 && N>=5)  A[1] = A[5];
S31 | if (i>=1 && N<=4)  B[i-1] = A[i];
S32 | if (i>=6 && N>=5)  B[i-6] = A[i-5];
```

$A_{S_1} = [1]$	$A_{S_{21}} = [1]$	$A_{S_{22}} = [1]$	$A_{S_{31}} = [1]$	$A_{S_{32}} = [1]$
$\beta_{S_1} = [2,0]$	$\beta_{S_{21}} = [2,0]$	$\beta_{S_{22}} = [2,0]$	$\beta_{S_{31}} = [2,0]$	$\beta_{S_{32}} = [2,0]$
$\Gamma_{S_1} = [0,0]$	$\Gamma_{S_{21}} = [0,0]$	$\Gamma_{S_{22}} = [0,5]$	$\Gamma_{S_{31}} = [0,1]$	$\Gamma_{S_{32}} = [0,6]$

Figure 3.45: Correcting + versioning S_3

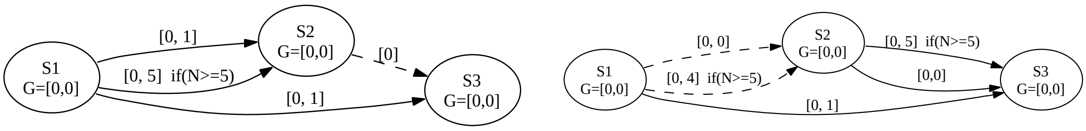


Figure 3.46: Outline of the correction for $p = 1$

Lemma 6 *Correction at a given depth by the minimal shift amount does not hamper correction at a subsequent depth.*

For $|p| > 1$, if VDG_p contains a negative weight cycle, $VDG_{|p|-1}$ contains a null weighted slackness cycle traversing the same nodes. By construction, any violated edge at depth p presupposes that the candidate violated polyhedron at previous depth $\delta_{\mathrm{v}|p|-1}^{S \to T}$ is not empty. Hence, for any violated edge at depth p, there exists a 0-slack edge at any previous depths thus ensuring the existence of a 0-slack cycle at any previous depths.□

In other words, the fact that a schedule cannot be corrected at a given depth is an intrinsic property of the schedule. Combined with the previous lemma, we deduce the completeness of our greedy algorithm.

Theorem 10 (Completeness) *If there exist shift amounts satisfying the system of violation and slackness constraints at all depths, then the correction algorithm removes all violations.*

Let us outline the algorithm on the example of Figure 3.40, assuming $N \geq 2$. Nodes represent statements of the program and are labeled with their respective schedules (β (or B) if $p \leq 0$ or Γ (or G) if $p > 0$). Dashed edges represent slackness while plain edges represent violations and are labeled with their constant or parametric amount. Suppose the chosen transformation tries to perform the modifications of the above statements' schedules according to the values in Figure 3.41.

The first pass of the correction algorithm for depth $p = 0$ detects the following loop independent violations: $S_1 \to S_2$, $S_1 \to S_2$ if $N \geq 5$, $S_1 \to S_3$, $S_2 \to S_3$ if $N \geq 5$. No slack edges are introduced in the initial graph. The violated dependence graph is a DAG and the correction mechanism will first push S_2 at the same β_0 as S_1. Then, after updating outgoing edges, it will do the same for S_3 yielding the code of Figure 3.42. Figure 3.43 shows the resulting VDG. So far, statement ordering within a given loop iteration is not fully specified (hence the statements are considered parallel); the code generation phase arbitrarily decides the shape of the final code. In addition, notice S_2 and S_3, initially located in different loop bodies than S_1, have been merged through the loop-independent step of the correction.

The next pass of the correction algorithm for depth $p = 1$ now detects the following loop carried violation: RAW dependence $\langle S_1, 5\rangle \rightarrow \langle S_2, 0\rangle$ is violated with the amount $5 - 0 = 5$ if $N \geq 5$, WAW dependence $\langle S_1, 1\rangle \rightarrow \langle S_2, 0\rangle$ is violated with the amount $1 - 0 = 1$, RAW dependence $\langle S_1, i+1\rangle \rightarrow \langle S_3, i\rangle$ is violated with the amount $i + 1 - i = 1$. There is also a 0-slack edge $S_2 \rightarrow S_3$ resulting from S_2 writing `A[1]` and S_3 reading it at iteration $i = 0$. All these informations are stored in the violation polyhedron. A maximization step with PIP determines a minimal shift amount of 5 if $N \geq 5$. Stopping the correction after shifting S_2 and versioning given the values of N would generate the intermediate result of Figure 3.44. However, since the versioning only happens at commit phases of the algorithm, the graph is given in Figure 3.46 and no node duplication is performed. The algorithm moves forward to correcting the node S_3 and, at step 3, the slack edge $S_2 \rightarrow S_3$ is updated with the new shift for S_2. The result is an incoming shift edge with violation amount 4 if $N >= 6$.

```
if (N<=4)
  A[0] = ...;
  A[1] = ...;
  A[1] = A[5];
  B[0] = A[1];
  for (i=2; i<=N; i++)
    A[i] = ...;
    B[i-1] = A[i];
  B[N-1] = A[N];
```

Figure 3.47: Case $N \leq 4$

```
else if (N>=5)
  for (i=0; i<=4; i++)
    A[i] = ...;
  A[5] = ...;
  A[1] = A[5];
  for (i=6; i<=N; i++)
    A[i] = ...;
    B[i-6] = A[i-5];
  for (i=N+1; i<=N+6; i++)
    B[i-6] = A[i-5];
```

Figure 3.48: Case $N \geq 5$

The separation and commit phases of our algorithm create as many disjoint versions of the correction as needed to enforce minimal shifts. The node duplications allow to express different schedules for different portions of the domain of each statement.

3.6.5 Correction by Index-Set Splitting

Index-set splitting has originally been crafted as an enabling transformation. It is usually formulated as a decision problem to express more parallelism by allowing the construction of piecewise affine functions. Yet, the iterative method proposed by Feautrier et al. [FGL99] relies on calls to a costly, non scalable, scheduling algorithm, and aims at exhibiting more parallelism by breaking cycles in the original dependence graph. However, significant portions of numerical programs do not exhibit such cycles, but still suffer from major inefficiencies; this is the case of the simplified excerpts from SPEC CPU2000fp benchmarks `swim` and `mgrid`, see Figures 3.51–3.49. Other methods fail to enable important optimizations in the presence of parallelization or fusion preventing dependences, or when loop bounds are not identical.

Feautrier's index-set splitting heuristic aims at improving the expressiveness of affine schedules. In the context of schedule corrections, the added expressiveness helps our greedy algorithm to find less intrusive (i.e., deeper) shifts. Nevertheless, since not all schedules may be corrected by a combination of translation and index-set splitting, it is interesting to have a local necessary criterion to rule out impossible solutions.

Correction Feasibility

When a VDG contains a circuit with negative weight, correction by translation alone becomes infeasible.

Notice that, if no circuit exists in the DG, then no circuit can exist in any VDG, since by construction, edges of the VDG are built from edges in the DG. In this case, whatever A, β and Γ parts are chosen for any statement, a correction is always found.

If the DG contains circuits, a transformation modifying β and Γ only is always correctable. Indeed, the reverse translation on β and Γ for all statements is a trivial admissible solution.

As a corollary, any combination of loop fusion, loop distribution and loop shifting [AK02] can be corrected. Thanks to this strong property of our algorithm, we can often completely eliminate the decision problem of finding enabling transformations such as loop bounds alignment, loop shifting and peeling.

In addition, this observation leads to a local necessary condition for ruling out non admissible correctable schedules.

Lemma 7 *Let $C = S \to S_1 \to \ldots \to S_n \to S$ be a circuit in the DG. By successive projections onto the image of every dependence polyhedron, we can incrementally construct a polyhedron $\delta^{S \to^P S_i}$ that contains all the instances of S and S_i that are transitively in dependence along the prefix P of the circuit. If $\delta^{S \to S}$ is not empty, the function $A^S_{p,\bullet} - A^S_{p,\bullet}$ from $\mathbb{Z}^{d_S+d_S+d_{gp}+1}$ to $\mathbb{Z}^{d_{gp}+1}$ must be positive for a correction to exist at depth p.*

Without this necessary property, index-set splitting would not enhance the expressiveness of affine schedules enough for a correction to be found (by loop shifting only). This is however not sufficient to ensure the schedule can be corrected.

Index-Set Splitting for Correction

Our splitting heuristic aims at preserving asymptotic locality and ordering properties of original shedule while avoiding code explosion. The heuristic runs along with the shifting-based correction algorithm, by splitting only target nodes of violated dependences. Intuitively, we decompose the target domain when the two following criteria hold:

1. the amount of correction is “too intrusive” with respect to the original (illegal) schedule;

2. it allows a “significant part” of the target domain to be preserved.

```
     for (i=0; i<N; i++)
S1   | A[i] = ...;
S2   A[0] = A[N-1];
     for (i=1; i<N; i++)
S3   | B[i] = A[i-1];
```

Figure 3.49: Original mgrid-like code

```
S1   A[0] = ...;
     for (i=1; i<N-1; i++)
S1   | A[i] = ...;
S31  | B[i+1] = A[i];
S1   A[N-1] = ...;
S2   A[0] = A[N-1];
S32  B[1] = A[0];
```

Figure 3.50: Optimized code

```
     for (i=0; i<N; i++)
     | for (j=0; j<N; j++)
S1   | | A[i][j] = ...;
     for (i=0; i<N; i++)
S2   | A[i][i] = ...;
     for (i=1; i<N; i++)
     | for (j=1; j<N; j++)
S3   | | B[i][j] = A[i][j];
```

Figure 3.51: Original swim-like code

```
     for (i=1; i<N; i++)
     | for (j=1; j<i-1; j++)
S1   | | A[i][j] = ...;
S3   | | B[i][j] = A[i][j];
S1   | A[i][i] = ...;
S2   | A[i][i] = ...;
S3   | B[i][i] = ...;
     | for (j=i+1; j<N; j++)
S1   | | A[i][j] = ...;
S3   | | B[i][j] = A[i][j];
```

Figure 3.52: Optimized code

Formally, a correction is deemed intrusive if it is a parametric shift (Γ) or a motion (β).

To assess the second criterion, we consider for every incoming shift edge, the projection of the polyhedron onto every non-parametric iterator dimension. A dimension whose projection is non-parametric and included in an interval smaller than a given constant (3 in our experiments) is called *degenerate*. In the following example, dimension j is degenerate:

$$\left\{\begin{array}{rcrcrcrcr}
-2i & + & 3j & + & 4M & + & 5 & \geq & 0 \\
i & + & j & - & M & + & 2 & \geq & 0 \\
i & - & 3j & + & 2M & - & 9 & \geq & 0 \\
-i & & & & & + & M & \geq & 0 \\
i & & & & & & & \geq & 0
\end{array}\right.$$

which simplifies into $2i+2 \geq 6j \geq 2i-5$.

To determine if a domain split should be performed, we incrementally remove violated parts of the target domain corresponding to intrusive corrections until either: we run out of incoming violations, or the remaining part of the domain is degenerate.

The intuition is to keep a non-degenerate core of the domain free of any parametric correction, to preserve locality properties of the original schedule. In the end, if the remaining portion of the domain still has the same dimension as the original one, a split is performed that separates the statement into the core that is not in violation and the rest of the domain.

Index set splitting is thus plugged into our correction algorithm as a preconditioning phase before step 4 of Figure 3.38.

Notice that a statement is only splitted a finite number of times, since each incoming shift edge is split at most once at each depth.

To limit the number of duplications, we allow only the core of a domain to be split among successive corrections. If a statement has already been decomposed at a given depth, only its core still exhibits the same locality properties as the original schedule. It is then unnecessary, and even harmful as far as code size is concerned, to further split the out-of-core part of the domain.

Figure 3.49 is a simplified version of one of the problems to solve when optimizing `mgrid`. The fusion of the first and third nests is clearly illegal since it would reverse the dependence from $\langle S_1, N-1\rangle$ to $\langle S_2\rangle$, as well as every dependence from $\langle S_1, i\rangle$ to $\langle S_3, i+1\rangle$. To enable this fusion, it is sufficient to shift the schedule of S_2 by $N-1$ iterations and to shift the schedule of S_3 by 1 iteration. Fortunately, only iteration $i=0$ of S_3 (after shifting by 1) is concerned by the violated dependence from S_2: peeling this iteration of S_3 gives rise to S_{3_1} and S_{3_2}. In turn, S_{3_1} is not concerned by the violation from S_2 while S_{3_2} must still be shifted by $N-1$ and eventually pops out of the loop. In the resulting code in Figure 3.50, the locality benefits of the fusion are preserved and the legality is ensured.

A simplified version of the `swim` benchmark exhibits the need for a more complex split. The code in Figure 3.51 features two doubly nested loops separated by an intermediate diagonal assignment loop, with poor temporal locality on array `A`. While allowing to maintain the order of the original schedule for a non-degenerate, triangular portion of S_1 and S_3 instances (i.e., $\{(i,j) \in [1,N] \mid j \neq i\}$); the code in Figure 3.52 also exhibits much more reuse opportunities and yields better performance.

While originally targeted at preserving locality properties of given schedules during correction, our method also succeeds at breaking cycles in a violated dependence graph. Consider the following example from Feautrier et *al.* [FGL99] given in Figure 3.53, A_{S_1} is set to [0]. Since source and target parts of the domain are disjoint, any transformation satisfies the previously defined feasibility criterion.

A self violation is detected provided iterator i statisfies $i \geq 0$ and $-2i-2N \geq 0$. This domain is not degenerate w.r.t. the original one, hence a split will occur for $\{i \leq N\}$, which breaks the self violated dependence and allows a correction at depth -1, yielding the alternate parallelized code.

```
     for (i=0; i<N; i++)
S1   | A[2*N - i] = A[i];
```

$\mathrm{A}_{S_1} = [1]$
$\beta_{S_1} = [0,0]$
$\Gamma_{S_1} = [0,0]$

```
     DOALL(i ∈ [0, N])
S11  | A[2*N - i] = A[i];
     DOALL(i ∈ [N+1, 2*N])
S12  | A[2*N - i] = A[i];
```

$\mathrm{A}_{S_{11}} = [0]$ $\quad \mathrm{A}_{S_{12}} = [0]$
$\beta_{S_{11}} = [0,0]$ $\quad \beta_{S_{12}} = [0,1]$
$\Gamma_{S_{11}} = [0,0]$ $\quad \Gamma_{S_{12}} = [0,0]$

Figure 3.53: Original and parallelized Code

3.6.6 Experimental Results

Programming languages are aimed at helping the programmer to write concise, human readable code, that executes many iterations of a given sets of statements. However, loop program constructs often result in bad temporal locality in the number of loop iterations between producers and consumers. When applied to current machines, a compiler needs to restructure these compact loop constructs, in an attempt to reduce the life span of values produced on hot paths. This may improve cache or register usage, but at a high cost in program complexity.

Our correction algorithm is applicable under many different scenarios (multidimensional affine schedules and dependence graphs), and we believe it is an important leap towards bridging the gap between the abstraction level of compact loop-based programs and performance on modern architectures.

To make its benefits more concrete, we apply it to one of the most important loop transformation for locality: loop fusion. It is often impeded by combinatorial decision problems such as shifting, index-set splitting and loop bounds alignment to remove so called fusion preventing edges.

To give an intuition on the kind of correction effort that is needed to exhibit unexploited locality, we provide meaningful numbers that summarize the corrections performed by our algorithm when applying aggressive loop fusion on full SPEC CPU2000fp programs `swim` and `mgrid`.

We start from inlined versions of the programs which represent 100% of the execution time for `swim` and 75% for `mgrid`. As a locality-enhancing heuristic, we try to apply loop fusion for all loops and at all loop levels. Since this transformation violates numerous dependences, our correction mechanism is applied on the resulting multidimensional affine schedules. Very simple examples derived from these experimentations have been shown in Figures 3.49–3.52. Both loops exhibit “hot statements” with important amounts of locality to be exploited after fusion. As indicated in Figure 3.54, `mgrid` has 3 hot statements in a 3-dimensional loop, but 12 statements are interleaved with these hot loops and exhibit dependences that prevent fusion at depth 2; `swim` exhibits 13 hot statements with 34 interleaved statements preventing fusion at depths 2 and 3.

Program	`mgrid`	`swim`
# source statements	31	99
# corrected statements	47	138
# source code size	88	132
# corrected code size	542	447
# hot statements	3	13
# fusion preventing statements	12	34
# peel	12	20
# triangular splits	0	5
# original distance	(4N,0,0) (8N,0,0)	3·(0,3N,0) 6·(0,5N,0)
# final distance	(2,1,0) (3,3,0)	11·(0,0,0) (0,1,0) (0,0,1)

Figure 3.54: Correction Experiments

Application of our greedy algorithm successfully results in the aggressive fusion of the compute cores, while inducing only small shifts. For `mgrid`, the hot statements once in different loop bodies — separated by distances $(4\times N,0,0)$ and $(8\times N,0,0)$ — are fused towards the innermost level with final translation vectors $(0,0,0)$ for the first, $(2,1,0)$ for the second and $(3,3,0)$ for the third one. For `swim`, the original statements once in separate doubly nested loops have been fused thanks to an intricate combination of triangular index-set splitting, peeling of the boundary iterations, and shifting; 11 statements required no shifting, 1 required $(0,1,0)$ and the other $(0,0,1)$.

Peeling and index-set splitting are required as to avoid loop distribution and are quantified in the 7^{th} and 8^{th} rows in Figure 3.54. Overall, the number of statements introduced by index set splitting is about $20-30$% which is quite reasonable.[11]

A performance evaluation was also conducted on a 2.4GHz AMD Athlon64 with 1MB L2 cache (3700+, running in x86_64 mode) and 3GB of DDR2 SDRAM. Compared to the **peak** SPEC CPU2000 figures,[12] the systematic fusion with automatic correction achieves a speed-up of **15%** for `swim`.

On the contrary, the same optimization applied to `mgrid` degrades performance by about 10%. This is not surprising: this benchmark is bound by memory bandwidth, and the 3D-stencil pattern of the loop nest requires about 27 registers to allow register reuse, too much for the x86_64 ISA. Also, no significant improvement can

[11] But code generation may induce a larger syntactic code size increase.

[12] I.e., with one the best compilers and the optimal combination of optimization flags.

be expected regarding L2-cache locality (on top of the back-end compiler's optimizations). The performance degradation results from the complexity of the nest after loop peeling and shifting. We were not able to optimize `mgrid` for the Intel Itanium 2 processor due to unstabilities in the Open64 compiler, but we expect strong speed-ups on this platform.[13]

No existing optimizing compiler is capable (up to our knowledge) of discovering the opportunity and applying such aggressive fusions. In addition, existing compilers deal with combinatorial decision problems associated with the selection of enabling transformations. All these decision problems disappear naturally with our correction scheme, in favor of more powerful heuristics that aim at limiting the amount of duplication in the resulting code while enforcing the compute intensive part of the program benefits from locality or parallelization improvements.

We are exploring the application of this correction algorithm to the compression of optimization search spaces, combining empirical heuristics (e.g., Monte-Carlo searches and machine-learning) with linear programming tools [PBCV07]. We are also using these results in the design of a meta-programming environment for loop transformations, where automatic correction allows to raise the level of abstraction for the expert programmer involved in the semi-automatic optimization of a program.

3.7 Related Work

The closest technical work has already been discussed in the previous sections. This section surveys the former efforts in designing an advanced loop nest transformation infrastructure and representation framework.

Most loop restructuring compilers introduced syntax-based models and intermediate representations. ParaScope [CHH$^+$93] and Polaris [BEF$^+$96] are dependence based, source-to-source parallelizers for Fortran. KAP [KAP] is closely related to these academic tools.

SUIF [H$^+$96] is a platform for implementing advanced compiler prototypes. SUIF was the vehicle for important advances in polyhedral compilation [LL97, LLL01], but the resulting prototypes had little impact due to a weak code generation method and the lack of scalability of the core algorithms. PIPS [IJT91] is one of the most complete loop restructuring compiler, implementing interprocedural polyhedral analyses and transformations (including an advanced array region analysis, automatic parallelization at the function and loop level, and a limited form of affine scheduling); it uses a syntax tree extended with polyhedral annotations, but not a unified polyhedral representation.

Closer to our motivations, the MARS compiler [O'B98] has been applied to iterative optimization [KKGO01]; this compiler's goal is to unify classical dependence-based loop transformations with data storage optimizations. However, the MARS intermediate representation only captures part of the loop-specific information (domains and access functions): it lacks the characterization of iteration orderings through multidimensional affine schedules. Recently, a similar unified representation has been applied to the optimization of compute-intensive Java programs, combining machine learning and iterative optimization [LO04]; again, despite the unification of multiple transformations, the lack of multidimensional affine schedules hampers the ability to perform long sequences of transformations and complicates the characterization and traversal of the search space, ultimately limiting performance improvements.

To date, the most thorough application of the polyhedral representation was the Petit dependence analyzer and loop restructuring tool [Kel96], based on the Omega library [KPR95]. It provides space-time mappings for iteration reordering, and it shares our emphasis on per-statement transformations, but it is intended as a research tool for small kernels only. Our representation — whose foundations were presented in [CGT04] — improves on the polyhedral representation proposed by [Kel96], and this explains how and why it is the first one that enables the composition of polyhedral generalizations of classical loop transformations, decoupled from any syntactic form of the program. We show how classical transformations like loop fusion or tiling can be composed in any order and generalized to imperfectly-nested loops with complex domains, without intermediate translation to a syntactic form (which leads to code size explosion). Eventually, we use a code generation technique suitable to a polyhedral representation that is again significantly more robust than the code generation proposed in the Omega library [Bas04, VBC06].

[13] We try to fix these issues before the camera-ready version.

3.8 Future Work

To avoid diminishing returns in tuning sequences of program transformations, we advocate for the collapse of multiple optimization phases into a single, unconventional, iterative search algorithm. This said, it does not bring any concrete hope of any simplification of the problem... except if, by construction, the search space we explore is much simpler than the cross-product of the search spaces of a sequence of optimisation phases.

We are still far from a general multi-purpose iterative optimization phase — if it exists — but we already made one step in that direction: we built a search space suitable for iterative traversal that encompasses *all legal program transformations in a particular class*. Technically, we considered the whole class of loop nest transformations that can be modeled as *one-dimensional schedules* [Fea92], a significant leap in model and search space complexity compared to state-of-the-art applications of iterative optimization [PBCV07]. This is only a preliminary step, but it will shape our future work in the area. Up to now, we made the following contributions:

- we statically construct the optimization space of all, arbitrarily complex, arbitrarily long sequences of loop transformations that can be expressed as one-dimensional affine schedules (using a polyhedral abstraction);
- this search space is built free of illegal and redundant transformation sequences, avoiding them altogether at the very source of the exploration;
- we demonstrate multiple orders of magnitude reduction in the size of the search space, compared to filtering-based approaches on loop transformation sequences or state-of-the-art affine schedule enumeration techniques;
- these smaller search spaces are amenable to fast-converging, mathematically founded operation research algorithms, allowing to compute the exact size of the space and to traverse it exhaustively;
- our approach is compatible with acceleration techniques for feedback-directed optimization, in particular on machine-learning techniques which focus the search to a narrow set of most promising transformations;
- our source-to-source transformation tool yields significant performance gains on top of a heavily tuned, aggressive optimizing compiler.

We are extending this approach to multi-dimensional schedules, using empirical and statistical sampling techniques to learn how good schedules distribute dependences across multiple time dimensions. Also we can only apply such radical techniques to small codes, we will make all efforts to scale those to real-size benchmarks, building on modular affine scheduling approaches, algorithmic and mathematical formulation improvements, and empirical decoupling of the constraints.

3.9 Conclusion

The ability to perform numerous compositions of program transformations is driven by the development of iterative optimization environments, and motivated through the manual optimization of standard numerical benchmarks. From these experiments, we show that current compilers are challenged by the complexity of aggressive loop optimization sequences. We believe that little improvements can be expected without redesigning the compilation infrastructure for compositionality and richer search space structure.

We presented a polyhedral framework that enables the composition of long sequences of program transformations. Coupled with a robust code generator, our method avoids the typical restrictions and code bloat of long compositions of program transformations. These techniques have been implemented in the Open64/ORC/EKOPath compiler and applied to the swim benchmark automatically. We have also shown that our framework opens up new directions for searching for complex transformation sequences for automatic or semi-automatic optimization or parallelization.

Chapter 4

Quality High Performance Systems

In this chapter, we attempt to reconcile performance considerations, as dictated by the physical limitations and the architecture of the hardware, with the classical qualities attributed to computer systems: high-level programming comfort, a certain level of predictability and efficiency, and the ability to check for or enforce specific structural properties (real time and resources). It is intended as an optimistic first step, motivating further research at the join point of high-performance computing and synchronous languages. Although we are currenly developing a more general "clocked" concurrency model, the ideas presented in this chapter are likely to stand as theoretical and motivational foundations for our future work in the area.

For the pessimistic reader, it may also be viewed as an isolated uphill battle against the bleak semantical future shaped by thread-level parallelism, whether associated with a shared memory model, or a message-based communication infrastructure, or inspired from transactional semantics of concurrent data bases.

In Section 4.1, we introduce and motivate our *n*-synchronous model through the presentation of a simple high-performance video application. Section 4.2 formalizes the concepts of periodic clocks and synchronizability. Section 4.3 is our main contribution: starting from a core synchronous language *a la* LUSTRE, it presents an associated calculus on periodic clocks and extends this calculus to combine streams with *n*-synchronizable clocks. Section 4.4 describes the semantics of *n*-synchronous process composition through translation to a strictly synchronous program, by automatically inserting buffers with minimal size. Section 4.5 discusses related work at the frontier between synchronous and asynchronous systems. We conclude in Section 4.6.

4.1 Motivation

This work may contribute to the design of a wide range of embedded systems, but we are primarily driven by video stream processing for high-definition TV [GPRN04]. The main algorithms deal with picture scaling, picture composition (picture-in-picture) and quality enhancement (including picture rate up-conversions; converting the frame rate of the displayed video, de-interlacing flat panel displays, sharpness improvement, color enhancement, etc.). Processing requires considerable resources and involves a variety of pipelined algorithms on multidimensional streams.

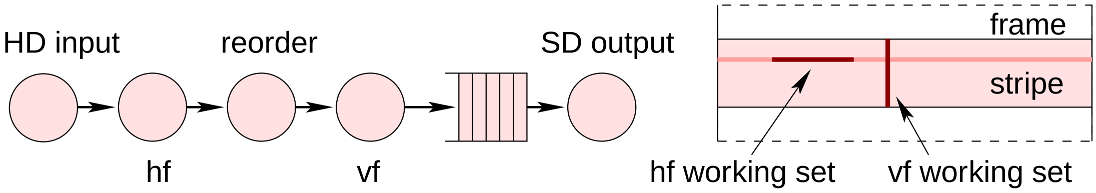


Figure 4.1: The downscaler

These applications involve a set of scalers that resize images in real-time. Our running example is a classical downscaler [CDC+03], depicted in Figure 4.1. It converts a high definition (HD) video signal, 1920 × 1080 pixels per frame, into a standard definition (SD) output for TV screen, that is 720 × 480:[1]

[1] Here we only consider the active pixels for the ATSC or BS-Digital High Definition standards.

1. A horizontal filter, hf, reduces the number of pixels in a line from 1920 down to 720 by interpolating packets of 6 pixels.

2. A reordering module, reorder, stores 6 lines of 720 pixels.

3. A vertical filter, vf, reduces the number of lines in a frame from 1080 down to 480 by interpolating packets of 6 pixels.

The processing of a given frame involves a constant number of operations on this frame only. A design tool is thus expected to automatically produce an efficient code for an embedded architecture, to check that real-time constraints are met, and to optimize the memory footprint of intermediate data and of the control code. The embedded system designer is looking for a programming language that offers precisely these features, and more precisely, which *statically* guarantees four important properties:

1. a proof that, according to worst-case execution time hypotheses, the frame and pixel rate will be sustained;

2. an evaluation of the delay introduced by the downscaler in the video processing chain, i.e., the delay before the output process starts receiving pixels;

3. a proof that the system has bounded memory requirements;

4. an evaluation of memory requirements, to store data within the processes, and to buffer the stream produced by the vertical filter in front of the output process.

In theory, synchronous languages are well suited to the implementation of the downscaler, enforcing bounded resource requirements and real-time execution. Yet, we show that existing synchronous languages make such an implementation tedious and error-prone.

4.1.1 The Need to Capture Periodic Execution

Technically, the scaling algorithm produces its t-th output (o_t) by interpolating 6 consecutive pixels (p_j) weighted by coefficients given in a predetermined matrix (example of a 64 phases, 6-taps polyphase filter [CDC$^+$03]):

$$o_t = \sum_{k=0}^{5} p_{t \times 1920/720+k} \times \mathrm{coef}(k, t \bmod 64).$$

```
let clock c = ok where rec
  cnt = 1 fby (if (cnt = 8) then 1 else cnt + 1)
  and ok = (cnt = 1) or (cnt = 3) or (cnt = 6)

let node hf p = o where rec
  o2 = 0 fby p and o3 = 0 fby o2 and o4 = 0 fby o3
  and o5 = 0 fby o4 and o6 = 0 fby o5
  and o = f (p,o2,o3,o4,o5,o6) when c

val hf :  int => int
val hf :: 'a -> 'a on c
```

Figure 4.2: Synchronous implementation of hf

Such filtering functions can easily be programmed in a strictly synchronous data-flow language such as LUSTRE or LUCID SYNCHRONE. Figure 4.2 shows a first version of the horizontal filter implemented in LUCID SYNCHRONE.

At every clock tick, the hf function computes the interpolation of six consecutive pixels of the input p (0 fby p stands for the previous value of p initialised with value 0). The implementation of f is out of the scope of this chapter; we will assume it sums its 6 arguments. The horizontal filter must match the production of 3 pixels for 8 input pixels. Moreover, the signal processing algorithm defines precisely the time when every pixel is emitted: the t-th output appears at the $t \times 1920/720$-th input. It can be factored in a periodic behavior of size 8, introducing an auxiliary boolean stream `c` used as a clock to sample the output of the horizontal filter. The `let clock` construction identifies syntactically these particular boolean streams. Here is a possible execution diagram.

`c`	*true*	*false*	*true*	*false*	*false*	*true*	*false*	...
`p`	3	4	7	5	6	10	12	...
`o2`	0	3	4	7	5	6	10	...
`o3`	0	0	3	4	7	5	6	...
`o4`	0	0	0	3	4	7	5	...
`o5`	0	0	0	0	3	4	7	...
`o6`	0	0	0	0	0	3	4	...
`o`	3		14			35		...

In the synchronous data-flow model, each variable/expression is characterized both by its stream of values and by its *clock*, relative to a global clock, called the base clock of the system. The clock of any expression e is an infinite boolean stream where *false* stands for the absence and *true* for the presence. E.g., if `x` is an integer stream variable, then `x+1` and `x` have the same clock. A synchronous process transforms an input clock into an output clock. This transformation is encoded in the process *clock signature* or *clock type*. Clocks signatures are relative to some clock variables. E.g., the clock signature of `hf` is $\forall\alpha.\alpha \rightarrow \alpha \text{ on } c$ (printed `'a -> 'a on c`) meaning that for any clock α, if input `p` has clock α, then the output is on a subclock $\alpha \text{ on } c$ defined by the instant where the boolean condition c is *true*.

In synchronous languages, clock conditions such as `c` can be arbitrarily complex boolean expressions, meaning that compilers make no hypothesis on them. Yet the applications we consider have a periodic behavior; thus a first simplification consists in enhancing the syntax and semantics with the notion of *periodic clocks*.

4.1.2 The Need for a Relaxed Approach

Real-time constraints on the filters are deduced from the frame rate: the input and output processes enforce that frames are sent and received at 30Hz. This means that HD pixels arrive at $30 \times 1920 \times 1080 = 62,208,000Hz$ — called the HD pixel clock — and SD pixels at $30 \times 720 \times 480 = 10,368,000Hz$ — called the SD pixel clock — i.e., 6 times slower. From these numbers, the designer would like to know that the delay before seeing the first output pixel is actually **12000 cycles** of the HD pixel clock, i.e., $192.915\mu s$, and that the minimal size of the buffer between the vertical filter and output process is **880 pixels**. (This is not the transposition buffer, whose size is defined in the specification.)

Synchronous languages typically offer such guarantees and static evaluations by forcing the programmer to make explicit the synchronous execution of the application. Nevertheless, the use of any synchronous language requires the designer to *explicitly implement* a synchronous code to buffer the outgoing pixels at the proper output rate and nothing helps him/her to *automatically* compute the values **12000** and **880**. Unfortunately, pixels are produced by the downscaler following a periodic but complex event clock. The synchronous code for the buffer handles the storage of each pending write from the vertical filter into a dedicated register, until the time for the output process to fetch this pixel is reached. Forcing the programmer to provide the synchronous buffer code is thus tedious and breaks modular composition. This scheme is even more complex if we include blanking periods [GPRN04].

In the following, we design a language that makes the computation of process latencies and buffer sizes automatic, using explicit periodic clocks. Technically, we define a relaxed clock-equivalence principle, called n-synchrony. A given clock ck_1 is *n-synchronizable* with another clock ck_2 if there exists a data-flow (causality) preserving way of making ck_1 synchronous with ck_2 applying a constant delay to ck_2 and inserting an intermediate size-n FIFO buffer. This principle is currently restricted to periodic clocks defined as periodic infinite binary words. This is different and independent from retiming [LS91], since neither ck_1 nor ck_2 are modified (besides the optional insertion of a constant delay); schedule choices associated with ck_1 and ck_2 are not impacted by the synchronization process.

We also define a relaxed synchronous functional programming language whose clock calculus accepts n-synchronous composition of operators. To this end, a type system underlying a strictly synchronous clock calculus is extended with two subtyping rules. Type inference follows an ad-hoc but complete procedure.

We show that every n-synchronous program can be transformed into a synchronous one (0-synchronous), replacing bounded buffers by some synchronous code.

4.2 Ultimately Periodic Clocks

This section introduces the formalism for reasoning about periodic clocks of infinite data streams.

4.2.1 Definitions and Notations

Infinite binary words are words of $(0+1)^\omega$. For the sake of simplicity, we will assume thereafter that every infinite binary word has an infinite number of 1s.

We are mostly interested in a subset of these words, called *infinite ultimately periodic binary words* or simply *infinite periodic binary words*, defined by the following grammar:

$$\begin{array}{lcl} w & ::= & u(v) \\ u & ::= & \varepsilon \mid 0 \mid 1 \mid 0.u \mid 1.u \\ v & ::= & 0 \mid 1 \mid 0.v \mid 1.v \end{array}$$

where $(v) = \lim_n v^n$ denotes the infinite repetition of *period* v, and u is a prefix of w. Let $\mathbb{Q}_2$ denote the set of infinite periodic binary words; it coincides with the set of rational diadic numbers [Vui94]. Since we always consider infinite periodic binary words with an infinite number of 1s, the period v contains at least one 1. This corresponds to removing the integer numbers from $\mathbb{Q}_2$ and considering only $\mathbb{Q}_2 - \mathbb{N}$.

Let $|w|$ denote the length of w. Let $|w|_1$ denote the number of 1s in w and $|w|_0$ the number of 0s in w. Let $w[n]$ denote the n-th letter of w for $n \in \mathbb{N}$ and $w[1..n]$ the prefix of length n of w.

There are an infinite number of representations for an infinite periodic binary word. Indeed, (0101) is equal to (01) and to $01(01)$. Fortunately, there exists a normal representation: it is the unique representation of the form $u(v)$ with the shortest prefix *and* with the shortest period.

Let $[w]_p$ denote the position of the p-th 1 in w. We have $[1.w]_1 = 1$, $[1.w]_p = [w]_{p-1} + 1$ if $p > 1$, and $[0.w]_p = [w]_p + 1$. Finally, let us define the *precedence* relation $\preceq$ by

$$w_1 \preceq w_2 \iff \forall p \geq 1, [w_1]_p \leq [w_2]_p.$$

E.g., $(10) \preceq (01) \preceq 0(01) \preceq (001)$. This relation is a *partial order* on infinite binary words. It abstracts the causality relation on stream computations, e.g., to check that outputs are produced before consumers request them as inputs.

We can also define the upper bound $w \sqcup w'$ and lower bound $w \sqcap w'$ of two infinite binary words with

$$\begin{array}{l} \forall p \geq 1, [w \sqcup w']_p = \max([w]_p, [w']_p) \\ \forall p \geq 1, [w \sqcap w']_p = \min([w]_p, [w']_p). \end{array}$$

E.g., $1(10) \sqcup (01) = (01)$ and $1(10) \sqcap (01) = 1(10)$; $(1001) \sqcup (0110) = (01)$ and $(1001) \sqcap (0110) = (10)$.

Proposition 1 *The set* $\big((0+1)^\omega, \preceq, \sqcup, \sqcap, \bot = (1), \top = (0)\big)$ *is a complete lattice.*

Notice $\top$ is indeed (0) since $[(0)]_p = \infty$ for all $p > 0$.[2]

Eventually, the following remark allows most operations on infinite periodic binary words to be computed on finite words.

Remark 1 *Considering two infinite periodic binary words,* $w = u(v)$ *and* $w' = u'(v')$*, one may transform these expressions into equivalent representatives* $a(b)$ *and* $a'(b')$ *satisfying one of the following conditions.*

1. *One may choose* a, a', b, *and* b' *with* $|a| = |a'| = max(|u|, |u'|)$ *and* $|b| = |b'| = lcm(|v|, |v'|)$ *where lcm stands for* least common multiple. *Indeed, assuming* $|u| \leq |u'|$, $p = |u'| - |u|$ *and* $n = lcm(|v|, |v'|)$: $w = u.v[1]\dots v[p].\big((v[p+1]\dots v[p+|v|])^{n/|v|}\big)$ *and* $w' = u'(v'^{n/|v'|})$. *E.g., words* $010(001100)$ *and* $10001(10)$ *can be rewritten into* $01000(110000)$ *and* $10001(101010)$.

2. *Likewise, one may obtain prefixes and suffixes with the same number of 1s:* $w = a(b)$ *and* $w' = a'(b')$ *with* $|a|_1 = |a'|_1 = max(|u|_1, |u'|_1)$ *and* $|b|_1 = |b'|_1 = lcm(|v|_1, |v'|_1)$. *Indeed, suppose* $|u|_1 \leq |u'|_1$, $|v|_1 \leq |v'|_1$, $p = |u'|_1 - |u|_1$, $r = [v]_p$, *and* $n = lcm(|v|_1, |v'|_1)$: $w = u.v[1]\dots v[r].\big((v[r+1]\dots v[r+|v|])^{n/|v|_1}\big)$ *and* $w' = u'.(v'^{n/|v'|_1})$. *E.g., the pair of words* $010(001100)$ *and* $10001(10)$ *become* $010001(100001)$ *and* $10001(1010)$.

[2] Yet the restriction of this lattice to $\mathbb{Q}_2$ is *not* complete, neither upwards nor downwards, even within $\mathbb{Q}_2 - \mathbb{N}$.

3. *Finally, one may write $w = a(b)$ and $w' = a'(b')$ with $|a|_1 = |a'|$ and $|b|_1 = |b'|$. Indeed, suppose $|u|_1 \leq |u'|$, $|v|_1 \leq |v'|$, $p = |u'|_1 - |u|$, $r = [v]_p$, and $n = lcm(|v|_1, |v'|)$: $w = u.v[1]\dots\ v[r].\big((v[r+1]\dots v[r+|v|])^{n/|v|_1}\big)$ and $w' = u'(v'^{n/|v'|})$. E.g., the pair of words* $010(001100)$ *and* $10001(10)$ *can be rewritten into* $0100011000011(000011)$ *and* $10001(10)$.

4.2.2 Clock Sampling and Periodic Clocks

A clock for infinite streams can be an infinite binary word or a composition of those, as defined by the following grammar:

$$c ::= w \mid c \ \mathtt{on}\ w, \quad w \in \{0,1\}^{\omega}.$$

If c is a clock and w is an infinite binary word, then $c \ \mathtt{on}\ w$ denotes a *subsampled clock* of c, where w is itself set on clock c. In other words, $c \ \mathtt{on}\ w$ is the clock obtained in advancing in clock w at the pace of clock c. E.g., $(01) \ \mathtt{on}\ (101) = (010101) \ \mathtt{on}\ (101) = (010001)$.

c	0	1	0	1	0	1	0	1	0	1	...	(01)
w		1		0		1		1		0	...	(101)
$c \ \mathtt{on}\ w$	0	1	0	0	0	1	0	1	0	0	...	(010001)

Formally, on is inductively defined as follows:

$$\begin{array}{rcl} 0.w \ \mathtt{on}\ w' & = & 0.(w \ \mathtt{on}\ w') \\ 1.w \ \mathtt{on}\ 0.w' & = & 0.(w \ \mathtt{on}\ w') \\ 1.w \ \mathtt{on}\ 1.w' & = & 1.(w \ \mathtt{on}\ w') \end{array}$$

Clearly, the on operator is *not* commutative.

Proposition 2 *Given two infinite binary words w and w', the infinite binary word $w \ \mathtt{on}\ w'$ satisfies the equation $[w \ \mathtt{on}\ w']_p = [w]_{[w']_p}$ for all $p \geq 1$.*

Proof. *This is proven by induction, observing that w' is traversed at the rate of 1s in w. $[w \ \mathtt{on}\ w']_1$ is associated with the q-th 1 of w such that q is the rank of the first 1 in w', i.e., $q = [w']_1$. Assuming the equation is true for p, the same argument proves that $[w \ \mathtt{on}\ w']_{p+1} = [w]_{[w']_p+q}$ where q is the distance to the next 1 in w', i.e., $q = [w']_{p+1} - [w']_p$, which concludes the proof.* □

There is an important corollary:

Proposition 3 (on-associativity) *Let w_1, w_2 and w_3 be three infinite binary words.*
Then $w_1 \ \mathtt{on}\ (w_2 \ \mathtt{on}\ w_3) = (w_1 \ \mathtt{on}\ w_2) \ \mathtt{on}\ w_3$.

Indeed $[w_1 \ \mathtt{on}\ w_2]_{[w_3]_p} = [w_1]_{[w_2]_{[w_3]_p}} = [w_1]_{[w_2 \ \mathtt{on}\ w_3]_p}$.

The following properties also derive from Proposition 2:

Proposition 4 (on-distributivity) *the on operator is distributive with respect to the lattice operations $\sqcap$ and $\sqcup$.*

Proposition 5 (on-monotonicity) *For any given infinite binary word w, functions $x \mapsto x \ \mathtt{on}\ w$ and $x \mapsto w \ \mathtt{on}\ x$ are monotone. The latter is also injective but* not *the former.*[3]

Using infinite binary words, we can exhibit an interesting set of clocks that we call *ultimately periodic clocks* or simply *periodic clocks*. A periodic clock is a clock whose stream is periodic. Periodic clocks are defined as follows:

$$c ::= w \mid c \ \mathtt{on}\ w, \quad w \in \mathbb{Q}_2.$$

In the case of these periodic clocks, proposition 2 becomes an algorithm, allowing to effectively compute the result of $c \ \mathtt{on}\ w$. Let us consider two infinite periodic binary words $w_1 = u_1(v_1)$ and $w_2 = u_2(v_2)$ with $|u_1|_1 = |u_2|$ and $|v_1|_1 = |v_2|$, this is possible because of Remark 1. Then $w_3 = w_1 \ \mathtt{on}\ w_2 = u_3(v_3)$ is computed by $|u_3| = |u_1|$, $|u_3|_1 = |u_2|_1$, $[u_3]_p = [u_1]_{[u_2]_p}$ and $|v_3| = |v_1|$, $|v_3|_1 = |v_2|_1$, $[v_3]_p = [v_1]_{[v_2]_p}$.

Likewise, periodic clocks are closed for the pointwise extensions of boolean operators or, not, and &.

[3] E.g., $(1001) \ \mathtt{on}\ (10) = (1100) \ \mathtt{on}\ (10)$.

4.2.3 Synchronizability

Motivated by the downscaler example, we introduce an equivalence relation to characterize the concept of resynchronization of infinite binary words (not necessarily periodic).

Definition 16 (synchronizable words) *We say that infinite binary words w and w' are* synchronizable, *and we write $w \bowtie w'$, iff there exists $d, d' \in \mathbb{N}$ such that $w' \preceq 0^d w'$ and $w' \preceq 0^{d'} w$. It means that we can delay w by d' ticks so that the 1s of w' occur before the 1s of w, and reciprocally.*

It means that the n-th 1 of w is at a bounded distance from the n-th 1 of w'. E.g., $1(10)$ and (01) are synchronizable; $11(0)$ and (0) are not synchronizable; (010) and (10) are not synchronizable since there are asymptotically too many reads or writes.

In the case of periodic clocks, the notion of synchronizability is computable.

Proposition 6 *Two infinite periodic binary words $w = u(v)$ and $w' = u'(v')$ are* synchronizable, *denoted by $w \bowtie w'$, iff they have the same* rate *(a.k.a.* throughput*)*

$$|v|_1/|v| = |v'|_1/|v'|.$$

In other words, $w \bowtie w'$ means w and w' have the same fraction of 1s in (v) and (v'), hence the same asymptotic production rate. It also means the n-th 1 of w is at a bounded distance from the n-th 1 of w'.

Proof. *From Remark 1, consider $w_1 = u(v)$ and $w_2 = u'(v')$ with $|u| = |u'|$ and $|v| = |v'|$. $w_1 = u(v) \bowtie w_2 = u'(v')$ iff there exists d, d' s.t. $\forall w \leq w_2[1..|u| + |v| + d]$, $w' \leq 0^d.w_2[1..|u| + |v|] \wedge |w| = |w'| \implies |w|_1 \geq |w'|_1$ and $\forall w \leq 0^{d'} w_1[1..|u| + |v|]$, $w' \leq w_2[1..|u| + |v| + d'] \wedge |w| = |w'| \implies |w|_1 \geq |w'|_1$. It is sufficient to cover the prefixes of finite length $\leq |u| + |v| + max(d + d')$.*

Case $|v'|_1 = 0$ is straightforward. Let us assume that $|v|_1/|v'|_1 > |v|/|v'|$ (the case $|v|_1/|v'|_1 < |v|/|v'|$ is symmetric). Because of Remark 1, it means $|v|_1/|v'|_1 > 1$. Then it entails that (v) and (v') are not synchronizable so as w_1 and w_2. Let us denote $a = |v|_1 - |v'|_1$, then v^n has na 1 more than v'^n. Thus $v^n \preceq 0^{f(n)} v'^n$ where $|v^n| \geq f(n) \geq na$ and $f(n)$ is minimal in the sense that $v^n \not\preceq 0^{f(n)-1} v'^n$. It entails that $(v) \preceq 0^{lim f(n)}(v')$ and thus there are not synchronizable.

Conversely, assume $|v|_1/|v'|_1 = |v|/|v'|$. Since u and u' are finite, we have $1^r u \preceq 0^p u'$ and $1^k u' \preceq 0^q u$ with $r = max(0, |u'|_1 - |u|_1)$, $k = max(0, |u|_1 - |u'|_1)$. (v), $p = \min\{l \mid l \leq |u| + r \wedge 1^r u \preceq 0^l u'\}$ and $q = \min\{l \mid l \leq |u'| + \wedge 1^k u' \preceq 0^l u\}$. (v') are also synchronizable, thus $(v) \preceq 0^m (v')$ and $(v') \preceq 0^n (v)$. Then $w_1 \preceq 0^{p+m+r|v|} w_2$ and $w_2 \preceq 0^{q+n+k|v'|} w_1$. There is an additional delay of $r|v|$ since each period v holds at least one 1. □

4.3 The Programming Language

This section introduces a simple data-flow functional language on infinite data streams. The semantics of this language has a strictly synchronous core, enforced by a so-called *clock calculus*, a type system to reject non synchronous programs, following [CP96, CP03]. Our main contribution is to extend this core with a *relaxed interpretation of synchrony*. This is obtained by extending the clock calculus so as to accept the composition of streams whose clocks are "almost equal". These program can in turn be automatically transformed into conventional synchronous programs by inserting buffer code at proper places.

4.3.1 A Synchronous Data-Flow Kernel

We introduce a core data-flow language on infinite streams. Its syntax derives from [CGHP04]. Expressions (e) are made of constant streams (i), variables (x), pairs (e, e), local definitions of functions or stream variables (e `where` $x = e$),[4] applications ($e(e)$), initialized delays (e `fby` e) and the following sampling functions: e `when` pe is the sampled stream of e on the periodic clock given by the value of pe, and `merge` is the combination operator of complementary streams (with opposite periodic clocks) in order to form a longer stream; `fst` and `snd` are the classical access functions; e `at` e is a clock constraint, asserting the first operand to be clocked at the rate of the second. As a syntactic sugar, e `whenot` pe is the sampled stream of e on the negation of the periodic clock pe.

A program is made of a sequence of declarations of stream functions (`let node` $f\ x = e$) and periodic clocks (`period` $p = pe$). E.g., `period` $half = (01)$ defines the half periodic clock (the alternating bit sequence) and this

[4] Corresponds to `let` x `=` e `in` e in ML.

clock can be used again to build an other one like $\texttt{period}\ quarter = half\ \texttt{on}\ half$. Periodic clocks can be combined with boolean operators. Note that clocks are *static* expressions which can be simplified at compile time into the normal form $u(v)$ of infinite periodic binary words.

$$\begin{array}{lll}
e & ::= & x \mid i \mid (e,e) \mid e\ \texttt{where}\ x = e \mid e(e) \mid op(e,e) \\
 & & \mid e\ \texttt{fby}\ e \mid e\ \texttt{when}\ pe \mid \texttt{merge}\ pe\ e\ e \\
 & & \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid e\ \texttt{at}\ e \\
d & ::= & \texttt{let node}\ f\ x = e \mid d; d \\
dp & ::= & \texttt{period}\ p = pe \mid dp; dp \\
pe & ::= & p \mid w \mid pe\ \texttt{on}\ pe \mid \texttt{not}\ pe \mid pe\ \texttt{or}\ pe \mid pe\ \&\ pe
\end{array}$$

We can easily program the downscaler in this language, as shown in Figure 4.3. The `main` function consists in composing the various filtering functions. Notation `o at (i when (100000))` is a constraint given by the programmer; it states that the output pixel `o` must be produced at some clock α `on` (100000), thus 6 times slower than the input clock α.

```
let period c = (10100100)
let node hf p = o where rec (...)
  and o = f (p,o2,o3,o4,o5,o6) when c

let node main i = o at (i when (100000)) where rec
  t = hf i
  and (i1,i2,i3,i4,i5,i6) = reorder t
  and o = vf (i1,i2,i3,i4,i5,i6)
```

Figure 4.3: Synchronous code using periodic clock

4.3.2 Synchronous Semantics

The (synchronous) denotational semantics of our core data-flow language is built on classical theory of synchronous languages [CGHP04]. Up to syntactic details, this is essentially the core LUSTRE language. Nonetheless, to ease the presentation, we have restricted sampling operations to apply to periodic clocks only (whereas any boolean sequence can be used to sample a stream in existing synchronous languages). Moreover, these periodic clocks are defined globally as constant values. These period expressions can in turn be automatically transformed into plain synchronous code or circuits (i.e., expressions from e) [Vui94].

This kernel can be statically typed with straightforward typing rules [CGHP04]; we will only consider clock types in the following. In the same way, we do not consider causality and initialization problems nor the rejection of recursive stream functions. These classical analyses apply directly to our core language and they are orthogonal to synchrony.

The compilation process takes two steps.

1. A *clock calculus* computes all constraints satisfied by every clock, as generated by a specific *type system*. These constraints are resolved through a *unification* procedure, to *infer* a periodic clock for each expression in the program. If there is no solution, we prove that some expressions do not have a periodic execution consistent with the rest of the program: the program is not synchronous, and therefore is rejected.

2. If a solution is found, the *code generation* step transforms the data-flow program into an imperative one (executable, OCaml, etc.) where all processes are synchronously executed according to their actual clock.

Clock Calculus

We propose a type system to generate the clock constraints. The goal of the clock calculus is to produce judgments of the form $P,H \vdash e : ct$ meaning that "the expression e has *clock type ct* in the environments of periods P and the environment H".

Clock types[5] are split into two categories, clock schemes (σ) quantified over a set of clock variables (α) and unquantified clock types (ct). A clock may be a functional clock ($ct \rightarrow ct$), a product ($ct \times ct$) or a stream clock

[5] We shall sometimes say *clock* instead of *clock type* when clear from context.

(ck). A stream clock may be a sampled clock (ck `on` pe) or a clock variable (α).

$$\begin{array}{lcl} \sigma & ::= & \forall\alpha_1,\ldots,\alpha_m.ct \\ ct & ::= & ct \rightarrow ct \mid ct \times ct \mid ck \\ ck & ::= & ck \ \mathtt{on}\ pe \mid \alpha \\ H & ::= & [x_1 : \sigma_1,\ldots,x_m : \sigma_m] \\ P & ::= & [p_1 : pe_1,\ldots,p_n : pe_n] \end{array}$$

The distinction between clock types (ct) and stream clock types (ck) should not surprise the reader. Indeed, whereas Kahn networks do not have clock types [Kah74], there is a clear distinction between a channel (which receives some clock type ck), a stream function (which receives some functional clock type $ct \rightarrow ct'$) and a pair expression (which receives some clock type $ct \times ct'$ meaning that the two expressions do not necessarily have synchronized values).

Clocks may be instantiated and generalized. This is a key feature, to achieve modularity of the analysis. E.g, the horizontal filter of the downscaler has clock scheme $\forall\alpha.\alpha \rightarrow \alpha\ \mathtt{on}\ (10100100)$; this means that, if the input has any clock α, then the output has some clock $\alpha\ \mathtt{on}\ (10100100)$. This clock type can in turn be instantiated in several ways, replacing α by more precise stream clock type (e.g., some sampled clock $\alpha'\ \mathtt{on}\ (01)$).

The rules for instantiating and generalizing a clock type are given below. $FV(ct)$ denotes the set of free clock variables in ct.

$$\begin{array}{lcl} ct'[\vec{ck}/\vec{\alpha}] & \leq & \forall\vec{\alpha}.ct' \\ fgen(ct) & = & \forall\alpha_1,\ldots,\alpha_m.ct \text{ where } \alpha_1,\ldots,\alpha_m = FV(ct) \end{array}$$

The inequality in the first rule stands for "being more precise than": it states that a clock scheme can be instantiated by replacing variables with clock expressions; In the second rule, $fgen(ct)$ returns a fully generalized clock type where every variable in ct is quantified universally.

When defining periods, we must take care that identifiers are already defined. If P is a period environment (i.e., a function from period names to periods), we shall simply write $P \vdash pe$ when every free name appearing in pe is defined in P.

The clocking rules defining the predicate $P,H \vdash e : ct$ are now given in Figure 4.4 and are discussed below.

- A constant stream may have any clock ck (rule (IM)).
- The clock of an identifier can be instantiated (rule (INST)).
- The inputs of imported primitives must all be on the same clock (rule (OP)).
- Rule (FBY) states that the clock of e_1 `fby` e_2 is the one of e_1 and e_2 (they must be identical).
- Rule (WHEN) states that the clock of e `when` pe is a sub-clock of the clock of e and we write it ck `on` pe. In doing so, we must check that pe is a valid periodic clock.
- Rule (MERGE) states an expression `merge` $pe\ e_1\ e_2$ is well clocked and on clock ck if e_1 is on clock ck `on` pe and e_2 is on clock the complementary clock ck `on not` pe.
- Rule (APP) is the classical typing rule of ML type systems.
- Rule (WHERE) is the rule for recursive definitions.
- Rules (PAIR), (FST) and (SND) are the rules for pairs.
- Rule (CTR) for the syntax e_1 `at` e_2 states that the clock associated to e_1 is imposed by the clock of e_2; it is the type constraint for clocks.
- Node declarations (rule (NODE)) are clocked as regular function definitions. We write $H,x : ct_1$ as the clock environment H extended with the association $x : ct_1$. Because node definitions only apply at top-level (and cannot be nested), we can generalize every variable appearing in the clock type.[6]
- Rules (PERIOD), (DEFH) and (DEFP) check that period and stream variables are well formed, i.e., names in period and stream expressions are first defined before being used.

[6] This is slightly simpler than the classical generalization rule of ML which must restrict the generalization to variables which do not appear free in the environment.

$$\text{(IM)}\quad P,H \vdash i : ck$$

$$\text{(INST)}\quad \frac{ct \leq H(x)}{P,H \vdash x : ct}$$

$$\text{(OP)}\quad \frac{P,H \vdash e_1 : ck \quad P,H \vdash e_2 : ck}{P,H \vdash op(e_1,e_2) : ck}$$

$$\text{(FBY)}\quad \frac{P,H \vdash e_1 : ck \quad P,H \vdash e_2 : ck}{P,H \vdash e_1 \ \mathtt{fby}\ e_2 : ck}$$

$$\text{(WHEN)}\quad \frac{P,H \vdash e : ck \quad P \vdash pe}{P,H \vdash e \ \mathtt{when}\ pe : ck \ \mathtt{on}\ pe}$$

$$\text{(MERGE)}\quad \frac{P \vdash pe \quad H \vdash e_1 : ck \ \mathtt{on}\ pe \quad P,H \vdash e_2 : ck \ \mathtt{on\ not}\ pe}{P,H \vdash \mathtt{merge}\ pe\ e_1\ e_2 : ck}$$

$$\text{(APP)}\quad \frac{P,H \vdash e_1 : ct_2 \rightarrow ct_1 \quad P,H \vdash e_2 : ct_2}{P,H \vdash e_1(e_2) : ct_1}$$

$$\text{(WHERE)}\quad \frac{P,H,x : ct_1 \vdash e_1 : ct_1 \quad P,\ H,x : ct_1 \vdash e_2 : ct_2}{P,H \vdash e_2 \ \mathtt{where}\ x = e_1 : ct_2}$$

$$\text{(PAIR)}\quad \frac{P,H \vdash e_1 : ct_1 \quad P,H \vdash e_2 : ct_2}{P,H \vdash (e_1,e_2) : ct_1 \times ct_2}$$

$$\text{(FST)}\quad \frac{P,H \vdash e : ct_1 \times ct_2}{P,H \vdash \mathtt{fst}\ e : ct_1}$$

$$\text{(SND)}\quad \frac{P,H \vdash e : ct_1 \times ct_2}{P,H \vdash \mathtt{snd}\ e : ct_2}$$

$$\text{(CTR)}\quad \frac{P,H \vdash e_1 : ck \quad P,H \vdash e_2 : ck}{P,H \vdash e_2 \ \mathtt{at}\ e_1 : ck}$$

$$\text{(NODE)}\quad \frac{P,\ H,x : ct_1 \vdash e : ct_2}{H \vdash \mathtt{let\ node}\ f\ x = e : [f : fgen(ct_1 \rightarrow ct_2)]}$$

$$\text{(PERIOD)}\quad \frac{P \vdash pe}{P \vdash \mathtt{period}\ p = pe : [p : pe]}$$

$$\text{(DEFH)}\quad \frac{H \vdash dh_1 : H_1 \quad H,H_1 \vdash dh_2 : H_2}{H \vdash dh_1; dh_2 : H_1,H_2}$$

$$\text{(DEFP)}\quad \frac{P \vdash dp_1 : P_1 \quad P,P_1 \vdash dp_2 : P_2}{P \vdash dp_1; dp_2 : P_1,P_2}$$

Figure 4.4: The core clock calculus

Structural Clock Unification

In synchronous data-flow languages such as LUSTRE or LUCID SYNCHRONE, clocks can be made of arbitrarily complex boolean expressions. In practice, the compiler makes no hypothesis on the condition c in the clock type $(ck\ \mathtt{on}\ c)$. This expressiveness is an essential feature of synchronous languages but forces the compiler to use a syntactical criteria during the unification process: two clock types $(ck_1\ \mathtt{on}\ c_1)$ and $(ck_2\ \mathtt{on}\ c_2)$ can be unified if ck_1 and ck_2 can be unified and if c_1 and c_2 are syntactically equal.

This approach can also be applied in the case of periodic clocks. Two clock types $(ck\ \mathtt{on}\ w_1)$ and $(ck_2\ \mathtt{on}\ w_2)$ can be unified if ck_1 and ck_2 can be unified and if $w_1 = w_2$ (for the equality between infinite binary words). As a result, this structural clock unification is unable to compare $(\alpha\ \mathtt{on}\ (01))\ \mathtt{on}\ (01)$ and $\alpha\ \mathtt{on}\ (0001)$ though two stream on these clocks are present and absent at the very same instants. A more clever unification mechanism will be the purpose of section 4.3.3.

Semantics over Clocked Streams

We provide our language with a data-flow semantics over finite and infinite sequences following Kahn formulation [Kah74]. Nonetheless, we restrict the Kahn semantics by making the absence of a value explicit. The set of instantaneous values is enriched with a special value $\perp$ representing the absence of a value.

We need a few preliminary notations. If T is a set, T^∞ denotes the set of finite or infinite sequences of elements over the set T ($T^\infty = T^* + T^\omega$). The empty sequence is noted ε and $x.s$ denotes the sequence whose head is x and tail is s. Let $\leq$ be the prefix order over sequences, i.e., $x \leq y$ if x is a prefix of y. The ordered set $D = (T^\infty, \leq)$ is a complete partial order (CPO). If D_1 and D_2 are CPOs, then $D_1 \times D_2$ is a CPO with the coordinate-wise order. $[D_1 \rightarrow D_2]$ as the set of continuous functions from D_1 to D_2 is also a CPO by taking the pointwise order. If f is a continuous mapping from D_1 to D_2, we shall write $\mathit{fix}(f) = \mathit{lim}_{n\rightarrow\infty} f^n(\varepsilon)$ for the smallest fix point of f (Kleene theorem). We define the set $\mathit{ClockedStream}(T)$ of *clocked sequences* as the set of finite and infinite sequences of elements over the set $T_\perp = T \cup \{\perp\}$.

$$\begin{array}{lcl} T_\perp & = & T \cup \{\perp\} \\ \mathit{ClockedStream}(T) & = & (T_\perp)^\infty \end{array}$$

A clocked sequence is made of present or absent values. We define the clock of a sequence s as a boolean sequence (without absent values) indicating when a value is present. For this purpose, we define the function *clock* from clocked sequences to boolean sequences:

$$\begin{array}{lcl} \mathit{clock}(\varepsilon) & = & \varepsilon \\ \mathit{clock}(\perp.s) & = & 0.\mathit{clock}(s) \\ \mathit{clock}(x.s) & = & 1.\mathit{clock}(s) \end{array}$$

We shall use the letter v for present values. Thus, $v.s$ denotes a stream whose first element is present and whose rest is s whereas $\perp.s$ denotes a stream whose first element is absent. The interpretation of basic primitives of the core language over clocked sequences is given in figure 4.5. We use the mark # to distinguish the syntactic construct (e.g., `fby`) from its interpretation as a stream transformer.

- The `const` primitive produces a constant stream from an immediate value. This primitive is polymorphic since it may produce a value (or not) according to the environment. For this reason, we add an extra argument giving its clock. Thus, $\texttt{const}^\#\ i\ c$ denotes a constant stream with stream clock c ($\mathit{clock}(\texttt{const}^\#\ i\ c) = c$).
- For a binary operator, the two operands must be synchronous (together present or together absent) and the purpose of the clock calculus is to ensure it statically (otherwise, some buffering is necessary).
- `fby` is the unitary delay: it “conses” the head of its first argument to its second one. The arguments and result of `fby` must be on the same clock. `fby` corresponds to a two-state machine: while both arguments are absent, it emits nothing and stays in its initial state ($\texttt{fby}^\#$). When both are present, it emits its first argument and enters the new state ($\texttt{fby1}^\#$) storing the previous value of its second argument. In this state, it emits a value every time its two arguments are present.
- The sampling operator expects two arguments on the same clock. The clock of the result depends on the boolean condition (c).
- The definition of `merge` states that one branch must be present when the other is absent.
- Note that $\texttt{not}^\#$ and $\texttt{on}^\#$ operate on boolean sequences only. The other boolean operations on clocks, e.g. `or` and `&`, follow the same principle.

It is easy to check that all these functions are continuous on clocked sequences.

Semantics is given to expressions which have passed the clock calculus ($\vdash$ judgments). We define the interpretation of clock types as the following:

$$\begin{array}{lcl} [\![ct_1 \rightarrow ct_2]\!]_P & = & [[\![ct_1]\!]_P \rightarrow [\![ct_2]\!]_P] \\ [\![ct_1 \times ct_2]\!]_P & = & [\![ct_1]\!]_P \times [\![ct_2]\!]_P \\ s \in [\![\forall\alpha_1,...,\alpha_n.ct]\!]_P & = & \forall ck_1,...,ck_n, s \in [\![ct[ck_1/\alpha_1,...,ck_n/\alpha_n]]\!]_P \\ s \in [\![ck]\!]_P & = & \mathit{clock}(s) \leq P(ck) \end{array}$$

$$
\begin{array}{lcl}
\texttt{const}^{\#}\ i\ 1.c & = & i.\texttt{const}^{\#}\ i\ c \\
\texttt{const}^{\#}\ i\ 0.c & = & \bot.\texttt{const}^{\#}\ i\ c \\[4pt]
op^{\#}(s_1,s_2) & = & \varepsilon \text{ if } s_1=\varepsilon \text{ or } s_2=\varepsilon \\
op^{\#}(\bot.s_1,\bot.s_2) & = & \bot.op^{\#}(s_1,s_2) \\
op^{\#}(v_1.s_1,v_2.s_2) & = & (v_1\, op\, v_2).op^{\#}(s_1,s_2) \\[4pt]
\texttt{fby}^{\#}(\varepsilon,s) & = & \varepsilon \\
\texttt{fby}^{\#}(\bot.s_1,\bot.s_2) & = & \bot.\texttt{fby}^{\#}(s_1,s_2) \\
\texttt{fby}^{\#}(v_1.s_1,v_2.s_2) & = & v_1.\texttt{fby1}^{\#}(v_2,s_1,s_2) \\
\texttt{fby1}^{\#}(v,\varepsilon,s) & = & \varepsilon \\
\texttt{fby1}^{\#}(v,\bot.s_1,\bot.s_2) & = & \bot.\texttt{fby1}^{\#}(v,s_1,s_2) \\
\texttt{fby1}^{\#}(v,v_1.s_1,v_2.s_2) & = & v.\texttt{fby1}^{\#}(v_2,s_1,s_2) \\[4pt]
\texttt{when}^{\#}(\varepsilon,c) & = & \varepsilon \\
\texttt{when}^{\#}(\bot.s,c) & = & \bot.\texttt{when}^{\#}(s,c) \\
\texttt{when}^{\#}(v.s,1.c) & = & v.\texttt{when}^{\#}(s,c) \\
\texttt{when}^{\#}(v.s,0.c) & = & \bot.\texttt{when}^{\#}(s,c) \\[4pt]
\texttt{merge}^{\#}(c,s_1,s_2) & = & \varepsilon \text{ if } s_1=\varepsilon \text{ or } s_2=\varepsilon \\
\texttt{merge}^{\#}(1.c,v.s_1,\bot.s_2) & = & v.\texttt{merge}^{\#}(c,s_1,s_2) \\
\texttt{merge}^{\#}(0.c,\bot.s_1,v.s_2) & = & v.\texttt{merge}^{\#}(c,s_1,s_2) \\[4pt]
\texttt{not}^{\#}1.c & = & 0.\texttt{not}^{\#}c \\
\texttt{not}^{\#}0.c & = & 1.\texttt{not}^{\#}c \\[4pt]
\texttt{on}^{\#}(1.c_1,1.c_2) & = & 1.\texttt{on}^{\#}(c_1,c_2) \\
\texttt{on}^{\#}(1.c_1,0.c_2) & = & 0.\texttt{on}^{\#}(c_1,c_2) \\
\texttt{on}^{\#}(0.c_1,c_2) & = & 0.\texttt{on}^{\#}(c_1,c_2)
\end{array}
$$

Figure 4.5: Semantics for the core primitives

$$
\begin{array}{rcl}
\llbracket P,H\vdash op(e_1,e_2):ck\rrbracket_{\rho} & = & op^{\#}(\llbracket P,H\vdash e_1:ck\rrbracket_{\rho},\llbracket P,H\vdash e_2:ck\rrbracket_{\rho}) \\
\llbracket P,H\vdash x:ct\rrbracket_{\rho} & = & \rho(x) \\
\llbracket P,H\vdash i:ck\rrbracket_{\rho} & = & i^{\#}\llbracket ck\rrbracket_{P} \\
\llbracket P,H\vdash e_1\ \texttt{fby}\ e_2:ck\rrbracket_{\rho} & = & \texttt{fby}^{\#}(\llbracket P,H\vdash e_1:ck\rrbracket_{\rho},\llbracket P,H\vdash e_2:ck\rrbracket_{\rho}) \\
\llbracket P,H\vdash e\ \texttt{when}\ pe:ck\ \texttt{on}\ pe\rrbracket_{\rho} & = & \texttt{when}^{\#}(\llbracket P,H\vdash e:ck\rrbracket_{\rho},P(pe)) \\
\llbracket P,H\vdash \texttt{merge}\ pe\ e_1\ e_2:ck\rrbracket_{\rho} & = & \texttt{merge}^{\#}(P(pe),\llbracket P,H\vdash e_1:ck\ \texttt{on}\ pe\rrbracket_{\rho},\llbracket P,H\vdash e_2:ck\ \texttt{on not}\ pe\rrbracket_{\rho}) \\
\llbracket P,H\vdash e_1(e_2):ct_2\rrbracket_{\rho} & = & (\llbracket P,H\vdash e_1:ct_1\rightarrow ct_2\rrbracket_{\rho})(\llbracket P,H\vdash e_2:ct_1\rrbracket_{\rho}) \\
\llbracket P,H\vdash e_1,e_2:ct_1\times ct_2\rrbracket_{\rho} & = & (\llbracket P,H\vdash e_1:ct_1\rrbracket_{\rho},\llbracket P,H\vdash e_2:ct_2\rrbracket_{\rho}) \\
\llbracket P,H\vdash \texttt{fst}\ s_1,s_2:ct_1\rrbracket_{\rho} & = & s_1 \text{ where } s_1,s_2=\llbracket P,H\vdash e:ct_1\times ct_2\rrbracket_{\rho} \\
\llbracket P,H\vdash \texttt{snd}\ s_1,s_2:ct_2\rrbracket_{\rho} & = & s_2 \text{ where } s_1,s_2=\llbracket P,H\vdash e:ct_1\times ct_2\rrbracket_{\rho} \\
\llbracket P,H\vdash e'\ \texttt{where}\ x=e:ct'\rrbracket_{\rho} & = & \llbracket P,\ H,x:ct\vdash e':ct'\rrbracket_{\rho[x^{\infty}/x]} \\
 & & \text{ where } x^{\infty}=\mathit{fix}(d\mapsto\llbracket P,\ H,x:ct\vdash e:ct\rrbracket_{\rho[d/x]}) \\
\llbracket P,H\vdash \texttt{let node}\ f(x)=e:fgen(ct_1\rightarrow ct_2)\rrbracket_{\rho} & = & [(d\mapsto\llbracket P,\ H,x:ct_1\vdash e:ct_2\rrbracket_{\rho[d/x]})/f] \\
\llbracket P,H\vdash e_1\ \texttt{at}\ e_2:ck\rrbracket_{\rho} & = & \llbracket P,H\vdash e_1:ck\rrbracket_{\rho}
\end{array}
$$

Figure 4.6: Data-flow semantics over clocked sequences

In order to take away causality problems (which are treated by some dedicated analysis in synchronous languages), $\llbracket ck\rrbracket_P$ contains all the streams whose clock is a prefix of the value of ck (and in particular the empty sequence ε). This way, an equation $x=x+1$ which is well clocked (since $P,H,x:ck\vdash x+1:ck$) but not causal (its smallest solution is ε) can receive a synchronous semantics.

For any period environment P, clock environment H and any assignment ρ (which maps variable names to values) such that $\rho(x)\in\llbracket H(x)\rrbracket_P$, the meaning of an expression is given by $\llbracket P,H\vdash e:ct\rrbracket_{\rho}$ such that $\llbracket P,H\vdash e:ct\rrbracket_{\rho}\in\llbracket ct\rrbracket_P$. The denotational semantics of the language is defined structurally in Figure 4.6.

Example

Let us illustrate these definitions on the downscaler in Figure 4.3.

1. Suppose that the input `i` has some clock type α_1.
2. The horizontal filter has the following signature, corresponding to the effective synchronous implementation of the process: $\alpha_2 \to \alpha_2 \texttt{ on } (10100100)$.
3. Between the horizontal filter and the vertical filter, the reorder process stores the 5 previous lines in a sliding window of size 5, but has no impact on the clock besides delaying the output until it receives 5 full lines, i.e., $5 \times 720 = 3600$ cycles. We shall give to the reorder proess the clock signature $\alpha_3 \to \alpha_3 \texttt{ on } 0^{3600}(1)$.
4. The vertical filter produces 4 pixels from 9 pixels repeatedly across the 720 pixels of a stripe (6 lines). Its signature (matching the process's synchronous implementation) is:

   $$\alpha_4 \;\to\; \alpha_4 \texttt{ on } (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720})$$

   To simplify the presentation, we will assume in manual computations that the unit of computation of the vertical filter is a line and not a pixel, hence replace 720 by 1 in the previous signature, yielding: $\alpha_4 \to \alpha_4 \texttt{ on } (101001001)$.
5. Finally, the designer has required that if the global input `i` is on clock α_1, then the clock of the output `o` should be $\alpha_1 \texttt{ on } (100000)$ — the 6 times subsampled input clock — tolerating an additional delay that must automatically be deduced from the clock calculus.

The composition of all 5 processes yield the type constraints $\alpha_1 = \alpha_2$, $\alpha_3 = \alpha_2 \texttt{ on } (10100100)$, and $\alpha_4 = \alpha_3 \texttt{ on } 0^{3600}(1)$. Finally, after replacing variables by their definitions, we get for the output `o` the following clock type:

$$((\alpha_1 \texttt{ on } (10100100)) \texttt{ on } 0^{3600}(1)) \texttt{ on } (101001001) = \alpha_1 \texttt{ on } 0^{9600}(100001000000010000000100).$$

Yet, the result is *not* equal to the clock constraint stating that `o` should have clock type $\alpha_1 \texttt{ on } (100000)$. The downscaler is thus rejected in a conventional synchronous calculus. This is the reason why we introduce the *relaxed* notion of *synchronizability*.

4.3.3 Relaxed Synchronous Semantics

The downscaler example highlights a fundamental problem with the embedding of video streaming applications in a synchronous programming model. The designer often has good reasons to apply a synchronous operator (e.g., the addition) on two channels with different clocks, or to compose two synchronous processes whose signatures do not match, or to impose a particular clock which does not match any solution of the constraints equations. Indeed, in many cases, the conflicting clocks may be "almost identical", i.e., they have the same asymptotic production rate. This advocates for a more relaxed interpretation of synchronism. Our main contribution is a clock calculus to accept the composition of clocks which are "almost identical", as defined by the structural extension of the synchronizability relation on infinite binary words to stream clocks:

Definition 17 (synchronizable clocks) *We say that two stream clocks $ck \texttt{ on } w$ and $ck \texttt{ on } w'$ are* synchronizable, *and we write $ck \texttt{ on } w \bowtie ck \texttt{ on } w'$, if and only if $w \bowtie w'$.*

Notice this definition does not directly extend to stream clocks with different variables.

Buffer Processes

When two processes communicate with synchronizable clocks, and when causality is preserved (i.e., writes precede or coincide with reads), one may effectively generate synchronous code for storing (the bounded number of) pending writes.

Consider two infinite binary words w and w' with $w \preceq w'$. A *buffer* $\texttt{buffer}_{w,w'}$ is a process with the clock type $\texttt{buffer}_{w,w'} : \forall\beta.\beta \texttt{ on } w \to \beta \texttt{ on } w'$ and with the data-flow semantics of an unbounded lossless FIFO channel [Kah74]. The existence of such an (a priori unbounded) buffer is guaranteed by the causality of the communication (writes occur at clock w that precede clock w'). We are only interested in buffers of *finite size* (a.k.a. bounded buffers), where the size of a buffer is the maximal number of pending writes it can accomodate while preserving the semantics of an unbounded lossless FIFO channel.

Proposition 7 *Consider two processes $f : ck \to \alpha \; \mathtt{on} \; w$ and $f' : \alpha \; \mathtt{on} \; w' \to ck'$, with $w \bowtie w'$ and $w \preceq w'$. There exists a buffer $\mathtt{buffer}_{w,w'} : \forall\beta.\beta \; \mathtt{on} \; w \to \beta \; \mathtt{on} \; w'$ such that $f' \circ \mathtt{buffer}_{w,w'} \circ f$ is a $(0-)$synchronous composition (with the unification $\alpha = \beta$).*

Proof. *A buffer of size n can be implemented with n data registers x_i and $2n+1$ clocks $(w_i)_{1 \le i \le n}$ and $(r_i)_{0 \le i \le n}$. Pending writes are stored in data registers: $w_i[j] = 1$ means that there is a pending write stored in x_i at cycle j. Clocks r_i determine the instants when the process associated with w' reads the data in x_i: $r_i[j] = 1$ means that the data in register x_i is read at cycle j. For a sequence of pushes and pops imposed by clocks w and w', the following case distinction simulates a FIFO on the x_i registers* statically *controlled through clocks w_i and r_i:*

`NOP`: $w[j] = 0$ **and** $w'[j] = 0$**.** *No operation affects the buffer, i.e., $r_i[j] = 0$, $w_i[j] = w_i[j-1]$; registers x_i are left unchanged.*

`PUSH`: $w[j] = 1$ **and** $w'[j] = 0$**.** *Some data is written into the buffer and stored in register x_1, all the data in the buffer being pushed from x_i into x_{i+1}. Thus $x_i = x_{i-1}$ and $x_1 = input$, $\forall i > 2, w_i[j] = w_{i-1}[j-1]$, $w_1[j] = 1$ and $r_i[j] = 0$.*

`POP`: $w[j] = 0$ **and** $w'[j] = 1$**.** *Let $p = \max(\{0\} \cup \{1 \le i \le n | w_i[j-1] = 1\})$. If p is zero, then no register stores any data at cycle j: input data must be bypassed directly to the output, crossing the wire clocked by r_0, setting $r_i[j] = 0$ for $i > 0$ and $r_0[j] = 1$, $w_i[j] = w_i[j-1]$. Conversely, if $p > 0$, $\forall i \neq p, r_i[j] = 0$, $r_p[j] = 1$, $\forall i \neq p, w_i[j] = w_i[j-1]$ and $w_p[j] = 0$. Registers x_i are left unchanged (notice this is not symmetric to the `PUSH` operation).*

`POP; PUSH`: $w[j] = 1$ **and** $w'[j] = 1$**.** *This case boils down to the implementation of a `POP` followed by a `PUSH`, as defined in the two previous cases.*

□

Assuming w and w' are periodic and have been written $w = u(v)$ and $w' = u'(v')$ under the lines of Remark 1, it is sufficient to conduct the previous simulation for $|u| + |v|$ cycles to compute periodic clocks w_i and r_i. This leads to an implementation in a plain $(0-)$synchronous language; yet this implementation is impractical because each clock w_i or r_i has a worst case quadratic size in the maximum of the periods of w and w' (from the application of remark 1), yielding cubic control space, memory usage and code size. This motivates the search for an alternative buffer implementation decoupling the memory management for the FIFO from the combinatorial control space; such an implementation is proposed in Section 4.4.2.

Relaxed Clock Calculus

Let us now modify the clock calculus in two ways:

1. a subtyping [Pie02] rule (SUB) is added to the clock calculus to permit the automatic insertion of a finite buffer in order to synchronize clocks;
2. rule (CTR) is modified into a subtyping rule to allow automatic insertion (and calculation) of a bounded delay.

The Subtyping Rule

Definition 18 *The relation <: is defined by*

$$w <: w' \iff w \bowtie w' \wedge w \preceq w'.$$

This is a partial order, and its restriction to equivalence classes for the synchronizability relation ($\bowtie$) forms a complete lattice.

We structurally extend this definition to stream clocks $ck \; \mathtt{on} \; w$ and $ck \; \mathtt{on} \; w'$ where $w <: w'$.[7]

Relation <: defines a subtyping rule (SUB) on stream clocks types:

$$\text{(SUB)} \; \frac{P,H \vdash e : ck \; \mathtt{on} \; w \quad w <: w'}{P,H \vdash e : ck \; \mathtt{on} \; w'}$$

[7] Yet this definition does not directly extends to stream clocks with different variables.

This is a standard subsumption rule, and all classical results on subtyping apply [Pie02].

The clock calculus defined in the previous section rejects expressions such as $x+y$ when the clocks of x and y cannot be unified. With rule (SUB), we can relax this calculus to allow an expression e with clock ck to be used “as if it had” clock ck' as soon as ck and ck' are *synchronizable* and causality is preserved.

E.g., the following program is rejected in the $(0-)$synchronous calculus since, assuming x has some clock α, $\alpha\ \mathtt{on}\ (01)$ cannot be unified with $\alpha\ \mathtt{on}\ 1(10)$.

$$\begin{array}{l}\mathtt{let\ node}\ f(x) = y\ \mathtt{where}\\ y = (x\ \mathtt{when}\ (01)) + (x\ \mathtt{when}\ 1(10))\end{array}$$

Let e_1 denote expression (`x when` (01)) and e_2 denote expression (`x when` $1(10)$), and let us generate the type constraints for each construct in the program:

1. (NODE): suppose that the signature of f is of form $f : \alpha \to \alpha'$;

2. (+): the addition expects two arguments with the same clocks;

3. (WHEN): we get $ck_1 = \alpha\ \mathtt{on}\ (01)$ for the clock of e_1 and $ck_2 = \alpha\ \mathtt{on}\ 1(10)$ for the clock of e_2;

4. (SUB): because (01) and $1(10)$ are synchronizable, the two clocks $ck_1 = \alpha\ \mathtt{on}\ (01)$ and $ck_2 = \alpha\ \mathtt{on}\ 1(10)$ can be resynchronized into $\alpha\ \mathtt{on}\ (01)$, since $(01) <: (01)$ and $1(10) <: (01)$.

The final signature is $f : \forall\alpha.\alpha \to \alpha\ \mathtt{on}\ (01)$.

Considering the downscaler example, this subtyping rule (alone) does not solve the clock conflict: the imposed clock first needs to be delayed to avoid starvation of the output process. This is the purpose of the following rule.

The Clock Constraint Rule The designer may impose the clock of certain expressions. Rule (CTR) is relaxed into the following subtyping rule:

$$\text{(CTR)}\ \frac{P,H \vdash e_1 : ck\ \mathtt{on}\ w_1 \quad P,H \vdash e_2 : ck\ \mathtt{on}\ w_2 \quad w_1 <: 0^d w_2}{P,H \vdash e_1\ \mathtt{at}\ e_2 : ck\ \mathtt{on}\ 0^d w_2}$$

Consider the previous example with the additional constraint that the output must have clock (1001).

$$\begin{array}{l}\mathtt{let\ node}\ f(x) = y\ \mathtt{at}\ (x\ \mathtt{when}\ (1001))\ \mathtt{where}\\ y = (x\ \mathtt{when}\ (01)) + (x\ \mathtt{when}\ 1(10))\end{array}$$

We previously computed that $(x\ \mathtt{when}\ (01)) + (x\ \mathtt{when}\ 1(10))$ has signature $\alpha\ \to\ \alpha\ \mathtt{on}\ (01)$, and (01) does not unify with (1001). Rule (CTR) yields

$$\frac{P,H \vdash y : a\ \mathtt{on}\ (01), x\ \mathtt{when}\ (1001) : a\ \mathtt{on}\ (1001) \quad (01) <: 0(1001)}{P,H \vdash y\ \mathtt{at}\ (x\ \mathtt{when}\ (1001)) : a\ \mathtt{on}\ 0(1001)}$$

Finally, $f : \forall\alpha.\alpha \to \alpha\ \mathtt{on}\ 0(1001)$. Indeed, one cycle delay is the minimum to allow synchronization with the imposed output clock.

Relaxed Clock Calculus Rules The predicate $P,H \vdash_s e : ct$ states that an expression e has clock ct in the period environment P and the clock environment H, *under the use of some synchronization mechanism*. Its definition extends the one of $P,H \vdash e : ct$ with the new rules in Figure 4.7. The axiom and all other rules are identical to the ones in Figure 4.4, using $\vdash_s$ judgments instead of $\vdash$.

Thus, starting from a standard clock calculus whose purpose is to reject non-synchronous program, we extend it with *subtyping* rules expressing that a stream produced on some clock ck_1 can be read on the clock ck_2 as soon as ck_1 can be synchronized into ck_2, using some buffering mechanism. By presenting the system in two steps, the additional expressiveness with respect to classical synchrony is made more precise.

$$\text{(SUB)}\ \frac{P,H \vdash_s e : ck \ \texttt{on}\ w_1 \quad w_1 <: w_2}{P,H \vdash_s e : ck \ \texttt{on}\ w_2}$$

$$\text{(CTR)}\ \frac{P,H \vdash_s e_1 : ck \ \texttt{on}\ w_1 \quad P,H \vdash_s e_2 : ck \ \texttt{on}\ w_2 \quad w_1 <: 0^d w_2}{P,H \vdash_s e_1 \ \texttt{at}\ e_2 : ck \ \texttt{on}\ 0^d w_2}$$

Figure 4.7: The relaxed clock calculus

Relaxed Synchrony and the `fby` Operator Notice `fby` is considered a length preserving function in dataflow networks, hence its clock scheme $\forall\alpha.\alpha \times \alpha \to \alpha$ in the 0-synchronous case, and despite it only needs its first argument at the very first instant. In the relaxed case, we could have chosen one of the following clock signatures: $\forall\alpha.\alpha \ \texttt{on}\ 1(0)e \times \alpha \to \alpha$, $\forall\alpha.\alpha \ \texttt{on}\ 1(0) \times \alpha \ \texttt{on}\ 0(1) \to \alpha$, or $\forall\alpha.\alpha \times \alpha \ \texttt{on}\ 0(1) \to \alpha$. The first two signatures require the first argument to be present at the very first instant only, which is overly restrictive in practice. The third signature is fully acceptable, with the observation that the original length-preserving signature can be reconstructed by applying the subtyping rule $\alpha \ \texttt{on}\ (1) <: \alpha \ \texttt{on}\ 0(1)$. This highlights the fact that the `fby` operator is a one-place buffer.

Construction of the System of Clock Constraints

The system of clock constraints is build from the systematic application of the core rules in Figure 4.4 and the relaxed calculus rules in Figure 4.7. All rules are syntax directed except (SUB) whose application is implicit at each (function or operator) composition.

Rule (CTR) is a special case: the clock constraint is built by computing a possible value for the delay d. This computation is syntax directed, and we always choose to minimize delay insertion: $\mathrm{delay}(w,w') = \min\{l \mid w \preceq 0^l w'\}$. When $w \preceq w'$, no delay is necessary. Note that in general, $\mathrm{delay}(w,w') \neq \mathrm{delay}(w',w)$.

Proposition 8 *The delay to synchronize an infinite periodic binary word w with an imposed infinite periodic binary word w' can be automatically computed by the formula*

$$delay(w,w') = \max(\max_p([w]_p - [w']_p), 0).$$

On periodic words, this delay is effectively computable thanks to Remark 1.

Proof. *Indeed, let $d = \max(\max_p([w]_p - [w']_p), 0)$ and $v = 0^d w'$ we have $w \preceq v$ since for all p, $[v]_p = d + [w']_p$. Moreover, d is minimal: suppose there exists p such that $d - 1 < [w]_p - [w']_p$, then $v' = 0^{d-1} w'$ satisfies $[v]_p = d - 1 + [w']_p < [w]_p$. Thus, $w \not\preceq v'$.* □

For the simplified downscaler, the minimal delay to resynchronize the vertical filter with the output process is 0^{9603}, since 9603 (clock cycles) is the minimal value of d such that $0^{9600}(100001000000010000000100) \preceq 0^d(100000)$. For the real downscaler (with fully developed vertical filter signature), we automatically computed that the minimal delay was **12000** to permit communication with the SD output.

Unification

We need a better unification procedure on clock types than the structural one (see Section 4.3.2), types to obtain an effective resolution algorithm for this system of constraints. In our case, a syntactic unification would unnecessarily reject many synchronous programs with periodic clocks. We propose a semi-interpreted unification that takes into account the semantics of periodic clocks. More precisely, the unification of two clock types ct and ct' can be purely structural on functional and pair types, where no simplification on periodic clocks can be applied, but it has to be aware of the properties of the sampling operator (`on`) when unifying stream clock types of the form $ck \ \texttt{on}\ w$ and $ck' \ \texttt{on}\ w'$. Two cases must be considered.

First of all, unifying $\alpha \ \texttt{on}\ w$ and $\alpha \ \texttt{on}\ w'$ returns true if and only if $w = w'$.

In the most general case, assume α and α' are clock variables (clocks can be normalised, thanks to the associativity of `on`). Equation $\alpha \ \texttt{on}\ w = \alpha' \ \texttt{on}\ w'$ *always has an infinite number of solutions*; these solutions generate an infinite number of different infinite binary words. Intuitively, a periodic sampling of w consists of the insertion of 0s in w, in a periodic manner. If $w \preceq w'$, it is always possible to delay the p-th 1 in w (resp. w') until the p-th 1 in w' (resp. w) through the insertion of 0s in α (resp. in α'). Let us define the subsampling relation $\leq_{SS}$, such that

$$a \leq_{SS} a' \iff \exists \alpha, a = \alpha \; \mathtt{on} \; a'.$$

Note that if $a \leq_{SS} a'$ then $a' \preceq a$, but the converse is not true: $(01) \preceq (0011)$ and there is no solution α such that $(0011) = \alpha \; \mathtt{on} \; (01)$.

Proposition 9 *Relation $\leq_{SS}$ is a partial order.*

Proof. *$\leq_{SS}$ is trivially reflexive and transitive. Antisymmetry holds because $\preceq$ is a partial order, and $a \leq_{SS} a'$ implies $a' \preceq a$.* □

In a typical unification scheme, one would like to replace the above type equation by "the most general clock type satisfying the equation". We will see that there is indeed a most general word m such that all common subsamples of w and w' are subsamples of m ($\leq_{SS}$ is an upper semi-lattice), yet the expression of $m = v \; \mathtt{on} \; w = v' \; \mathtt{on} \; w'$ does not lead to a unique choice for m and for the maximal unifiers v and v'. In fact, there can be an infinite set of such words.

In a strictly synchronous setting, we need to fall back to an incomplete unification scheme (some synchronous programs with periodic clocks will be rejected), choosing one of these solutions. If (v, v') is the chosen solution, the unification of $a \; \mathtt{on} \; w$ and $a' \; \mathtt{on} \; w'$ yields a unique clock type $\alpha \; \mathtt{on} \; v \; \mathtt{on} \; w = \alpha \; \mathtt{on} \; v' \; \mathtt{on} \; w'$, and every occurence of a (resp. a') is replaced by $\alpha \; \mathtt{on} \; v$ (resp. $\alpha \; \mathtt{on} \; v'$).

Yet in our relaxed synchronous setting, the most general unifier has an interesting property:

Proposition 10 (synchronizable unifiers) *Consider w, v_1, v_2 such that $v_1 \; \mathtt{on} \; w = v_2 \; \mathtt{on} \; w$; we have $v_1 \bowtie v_2$.*

This directly derives from Proposition 2.

We may thus make an arbitrary choice for (v, v') among maximal unifiers, and select one that is easy to compute. Formally, we define the *earliest* substitutions $\mathcal{V}$ and $\mathcal{V}'$ through the following recurrent equations:

$$\begin{array}{rcl}
\mathcal{V}(0^d 1.w, 0^d 0^{d'} 1.w') & = & 1^d 0^{d'} 1.\mathcal{V}(w, w') \\
\mathcal{V}(0^d 0^{d'} 1.w, 0^d 1.w') & = & 1^d 1^{d'} 1.\mathcal{V}(w, w') \\
\mathcal{V}'(0^d 1.w, 0^d 0^{d'} 1.w') & = & 1^d 1^{d'} 1.\mathcal{V}'(w, w') \\
\mathcal{V}'(0^d 0^{d'} 1.w, 0^d 1.w') & = & 1^d 0^{d'} 1.\mathcal{V}'(w, w')
\end{array}$$

Let $\mathcal{M}(w, w')$ denote the unifier

$$\mathcal{M}(w, w') = \mathcal{V}(w, w') \; \mathtt{on} \; w = \mathcal{V}'(w, w') \; \mathtt{on} \; w'.$$

The computation of $\mathcal{V}$ and $\mathcal{V}'$ terminates on periodic words because there are a finite number of configurations (bounded by the product of the period lenghts of w and w').

E.g., $a \; \mathtt{on} \; (1000) = a' \; \mathtt{on} \; 0(101)$:

w	1 0 0 0 1 0 0 0 1 0 0 0 1 ...	(1000)
w'	0 1 0 1 1 0 1 1 0 1 1 0 1 ...	0(101)
$\mathcal{V}(w, w')$	0 1 1 1 1 1 1 1 1 1 1 1 1 ...	0(1)
$\mathcal{V}'(w, w')$	1 1 1 0 0 1 0 0 0 1 1 0 0 ...	1(11001000)
$\mathcal{M}(w, w')$	0 1 0 0 0 1 0 0 0 1 0 0 0 ...	0(1000)

Proposition 11 *For all w, w', p,*

$$[\mathcal{M}(w, w')]_{p+1} = [\mathcal{M}(w, w')]_p + \max([w]_{p+1} - [w]_p, [w']_{p+1} - [w']_p).$$

Proof. *An inductive proof derives naturally from the previous algorithm. In particular, observe that between two consecutive 1s in $\mathcal{M}(w, w')$, the associated subword of either v or v' is a sequence of 1s; hence either $[\mathcal{M}(w, w')]_{p+1} - [\mathcal{M}(w, w')]_p = [w]_{p+1} - [w]_p$ or $[\mathcal{M}(w, w')]_{p+1} - [\mathcal{M}(w, w')]_p = [w']_{p+1} - [w']_p$.* □

In addition, $\mathcal{M}(w, w')$ is *the maximum* common subsample of w and w' and has several interesting properties:

Theorem 11 (structure of subsamples) *The subsampling relation $\leq_{SS}$ forms an upper semi-lattice on infinite binary words, the supremum of a pair of words w, w' being $\mathcal{M}(w,w')$.*

Common subsamples of w and w' form a complete lower semi-lattice structure for $\preceq$, $\mathcal{M}(w,w')$ being the bottom element.

$\mathcal{M}$ is also associative: $\mathcal{M}(w,\mathcal{M}(w',w'')) = \mathcal{M}(\mathcal{M}(w,w'),w'')$. (Hence the complete lower semi-lattice structure for $\preceq$ holds for common subsamples of any finite set of infinite binary words.)

Proof. *We proceed by induction on the position of the p-th 1. Consider a infinite binary word $m' = u \mathsf{~on~} w = u' \mathsf{~on~} w'$. By construction of m, $[m]_1 = \max([w]_1,[w']_1)$, hence $[m]_1 \leq [m']_1$. Assume all common subsamples of w and w' are subsamples of m up to their p-th 1, and that $[m]_p \leq [m']_p$ for some $p \geq 1$. Proposition 11 tells that m is identical to either w or w' between its p-th and $p+1$-th 1; hence common subsamples of w and w' are subsamples of m up to the next 1; and since $w \preceq m'$ and $w' \preceq m'$ ($\leq_{SS}$ is a reversed sub-order of $\preceq$), we get $[m]_{p+1} \leq [m']_{p+1}$, hence $m \preceq m'$ by induction on p.*

Associativity derives directly from Proposition 11. □

Resolution of the System of Clock Constraints

We may now define a resolution procedure through a set of constraint-simplification rules.

$$\text{(CYCLE)} \quad S + \{\alpha \mathsf{~on~} w_1 <: \alpha \mathsf{~on~} w_2\} \rightsquigarrow S \quad \text{if } w_1 <: w_2$$

$$\text{(SUP)} \quad S + \{\alpha \mathsf{~on~} w_1 <: \alpha',\ \alpha \mathsf{~on~} w_2 <: \alpha'\} \rightsquigarrow S + \{\alpha \mathsf{~on~} w_1 \sqcup w_2 <: \alpha'\} \quad \text{if } w_1 \bowtie w_2$$

$$\text{(INF)} \quad S + \{\alpha' <: \alpha \mathsf{~on~} w_1,\ \alpha' <: \alpha \mathsf{~on~} w_2\} \rightsquigarrow S + \{\alpha' <: \alpha \mathsf{~on~} w_1 \sqcap w_2\} \quad \text{if } w_1 \bowtie w_2$$

$$\text{(EQUAL)} \quad S \rightsquigarrow S\begin{bmatrix} \alpha'_1 \mathsf{~on~} v_1/\alpha_1 \\ \alpha'_2 \mathsf{~on~} v_2/\alpha_2 \end{bmatrix} \quad \text{if} \quad \begin{array}{l} S = S' + I_1 + I_2, \text{ with} \\ I_1 = \{\alpha_1 \mathsf{~on~} w_1 <: ck_1\} \text{ or } \{ck_1 <: \alpha_1 \mathsf{~on~} w_1\}, \\ I_2 = \{\alpha_2 \mathsf{~on~} w_2 <: ck_2\} \text{ or } \{ck_2 <: \alpha_2 \mathsf{~on~} w_2\} \end{array} \begin{array}{l} \alpha_1 \neq \alpha_2 \\ w_1 \neq w_2 \end{array}, \begin{array}{l} v_1 = \mathcal{V}(w_1,w_2) \\ v_2 = \mathcal{V}'(w_1,w_2) \end{array}$$

$$\text{(CUT)} \quad S + \{\alpha_1 \mathsf{~on~} w <: \alpha_2 \mathsf{~on~} w\} \rightsquigarrow S + \{\alpha_1 <: \alpha_3 \mathsf{~on~} u_1,\ \alpha_3 \mathsf{~on~} u_2 <: \alpha_2\} \quad \text{if } \alpha_1 \neq \alpha_2,\ u_1 = \mathcal{U}_{\max}(w),\ u_2 = \mathcal{U}_{\min}(w)$$

$$\text{(FORK)} \quad S + \{\alpha <: \alpha_1 \mathsf{~on~} w,\ \alpha <: \alpha_2 \mathsf{~on~} w\} \rightsquigarrow S[\alpha_3 \mathsf{~on~} u \mathsf{~on~} w/\alpha] + \{\alpha_3 \mathsf{~on~} u <: \alpha_1,\ \alpha_3 \mathsf{~on~} u <: \alpha_2\} \quad \text{if } \alpha_1 \neq \alpha_2,\ u = \mathcal{U}_{\min}(w)$$

$$\text{(JOIN)} \quad S + \{\alpha_1 \mathsf{~on~} w <: \alpha,\ \alpha_2 \mathsf{~on~} w <: \alpha\} \rightsquigarrow S[\alpha_3 \mathsf{~on~} u \mathsf{~on~} w/\alpha] + \{\alpha_1 <: \alpha_3 \mathsf{~on~} u,\ \alpha_2 <: \alpha_3 \mathsf{~on~} u\} \quad \text{if } \alpha_1 \neq \alpha_2,\ u = \mathcal{U}_{\max}(w)$$

$$\text{(SUBST)} \quad S \oplus I \rightsquigarrow S[ck/\alpha] \quad \text{if } I = \{\alpha <: ck\} \text{ or } \{ck <: \alpha\},\ \alpha \notin FV(ck)$$

Figure 4.8: Clock constraints resolution

The clock system given is turned into an algorithm by introducing a subtyping rule at every application point and by solving a set of constraints of the form $ck_i <: ck'_i$. The program is well clocked if the set of constraints is satisfiable.

Definition 19 (constraints and satisfiability) *A system S of clock constraints is a collection of inequations between clock types:*

$$S \quad ::= \quad \{ck_1 <: ck'_1, \ldots, ck_n <: ck'_n\}$$

We write $S + \{ck_1 <: ck_2\}$ for the extension of a system S with the inequation $\{ck_1 <: ck_2\}$. We write $S \oplus \{ck_1 <: ck_2\}$ for $S + \{ck_1 <: ck_2\}$ such that S does not contain a directed chain *of inequations from any free variable in ck_1 to any free variable in ck_2. For example, $S \oplus \{\alpha_1 <: \alpha_2 \mathsf{~on~} w_2\}$ means that, in S, α_1 never appear on the left of an inequation that leads transitively to an inequation where α_2 appears on the right.*

A system S is satisfiable if there exists a substitution ρ from variables to infinite binary words such that for all $\{ck_i <: ck'_i\} \in S, \rho(ck_i) <: \rho(ck'_i)$.

There is a straightforward but important (weak) confluence property on subsampling and satisfiability:

Proposition 12 (subsampling and satisfiability) *If $\alpha' \notin S$, then for all w, S is satisfiable iff $S[\alpha' \mathsf{~on~} w/\alpha]$ is satisfiable.*

Proof. *Suppose S is satisfiable with $\rho(\alpha) = \gamma \mathsf{~on~} m$. Then we can build another substitution ρ' satisfying the system of constraints by choosing $\rho'(\gamma) = \gamma' \mathsf{~on~} \mathcal{V}(m,w)$, $\rho'(\alpha) = \gamma' \mathsf{~on~} \mathcal{V}(m,w) \mathsf{~on~} m$ and $\rho'(\alpha') = \gamma' \mathsf{~on~} \mathcal{V}'(m,w)$. The reciprocal is obvious.* □

Let us eventually define three functions useful to bound the set of subsamples of a given word: $\mathcal{U}_{\min}$, $\mathcal{U}_{\max}$ and Δ are defined recursively as follows:

$$\begin{array}{lcl} \mathcal{U}_{\min}(0^a1^b.w) & = & 1^a0^a0^b1^b.\mathcal{U}_{\min}(w) \\ \mathcal{U}_{\max}(0^a1^b.w) & = & 0^a0^b1^a1^b.\mathcal{U}_{\max}(w) \\ \Delta(u_1.u_2.u, 0^a1^b.w) & = & 1^a0^{c_1}1^a1^b.u_2.\Delta(u,w) \\ \quad \text{with } |u_1|_1 = 2a+b, |u_2|_1 = b, |u_1|_0 = c_1, |u_2|_0 = c_2 \end{array}$$

Notice Δ — from pairs of infinite binary words to infinite binary words— is of technical interest for the proofs only.

Proposition 13 *For all w, $\mathcal{U}_{min}(w) \bowtie \mathcal{U}_{max}(w)$, $\mathcal{U}_{min}(w) <: \mathcal{U}_{max}(w)$, and $\mathcal{U}_{min}(w) \text{ on } w = \mathcal{U}_{max}(w) \text{ on } w$.*

For all u, w, $\Delta(u,w)$ is an infinite periodic binary word and is synchronizable with u.

Proof. *The first part of the proposition is proven inductively on the position of 1s in the subsampling.*

The second part derives from $\frac{|1^a0^{c_1}1^a1^b.u_2|_1}{|1^a0^{c_1}1^a1^b.u_2|} = \frac{2a+2b}{2a+2b+c_1+c_2} = \frac{|u_1.u_2|_1}{|u_1.u_2|}$, where $u_1.u_2$ satisfies the constraints in the inductive definition of Δ. □

Proposition 14 *For all u, v, w,*

$$\Delta(u,w) \text{ on } \mathcal{U}_{min}(w) \text{ on } w = u \text{ on } \mathcal{U}_{min}(w) \text{ on } w =$$
$$u \text{ on } \mathcal{U}_{max}(w) \text{ on } w = \Delta(u,w) \text{ on } \mathcal{U}_{max}(w) \text{ on } w$$

Proof. *Since $\mathcal{U}_{min}(w) \text{ on } w = \mathcal{U}_{max}(w) \text{ on } w$, we have*
$u \text{ on } \mathcal{U}_{min}(w) \text{ on } w = u \text{ on } \mathcal{U}_{max}(w) \text{ on } w$ and
$\Delta(u,w) \text{ on } \mathcal{U}_{min}(w) \text{ on } w = \Delta(u,w) \text{ on } \mathcal{U}_{max}(w) \text{ on } w$ In addition, $\mathcal{U}_{min}(0^a1^b.w) \text{ on } 0^a1^b.w = 0^a0^b0^a1^b$, hence
$\Delta(u_1.u_2.u, 0^a1^b.w) \text{ on } \mathcal{U}_{min}(0^a1^b.w) \text{ on } 0^a1^b.w =$
$0^a0^{c_1}0^b0^a.u_2.\Delta(u,w) \text{ on } \mathcal{U}_{min}(w) \text{ on } w$ (since $|u_2|_1 = b$). Finally, $u_1.u_2.u \text{ on } \mathcal{U}_{min}(0^a1^b.w) \text{ on } 0^a1^b.w =$
$0^{2a+b+c_1}.u_2.u \text{ on } \mathcal{U}_{min}(w) \text{ on } w$. □

Proposition 15 *$\Delta(u,w) \text{ on } \mathcal{U}_{min}(w)$ is the minimum (for $\preceq$) of all v' such that*

$$v' \text{ on } w = \Delta(u,w) \text{ on } \mathcal{U}_{min}(w) \text{ on } w.$$

$\Delta(u,w) \text{ on } \mathcal{U}_{max}(w)$ is the maximum (for $\preceq$) of all v' such that

$$v' \text{ on } w = \Delta(u,w) \text{ on } \mathcal{U}_{max}(w) \text{ on } w.$$

Proof. *The result is proven by induction, observing that the p-th 1 in v' is enclosed between the p-th 1 in $\Delta(u,w) \text{ on } \mathcal{U}_{min}(w)$ and in $\Delta(u,w) \text{ on } \mathcal{U}_{max}(w)$. Indeed, we have $\Delta(u_1.u_2.u, 0^a1^b.w) \text{ on } \mathcal{U}_{min}(0^a1^b.w) =$ $1^a0^{c_1}0^a0^b.u_2.\Delta(u,w) \text{ on } \mathcal{U}_{min}(w)$, and $\Delta(u_1.u_2.u, 0^a1^b.w) \text{ on } \mathcal{U}_{max}(0^a1^b.w) = 0^a0^{c_1}0^b1^a.u_2.\Delta(u,w) \text{ on } \mathcal{U}_{max}(w)$.* □

The set of subsamples of a given word is characterized by the following technical proposition:

Proposition 16 *For all u, v, w,*

$$u \text{ on } \mathcal{U}_{min}(w) \text{ on } w <: v \text{ on } w \implies \Delta(u,w) \text{ on } \mathcal{U}_{min}(w) <: v$$

and

$$v \text{ on } w <: u \text{ on } \mathcal{U}_{max}(w) \text{ on } w \implies v <: \Delta(u,w) \text{ on } \mathcal{U}_{max}(w).$$

Proof. *Observe that $u \text{ on } \mathcal{U}_{min}(w) \text{ on } w <: v \text{ on } w$. From Proposition 14, we have $\Delta(u,w) \text{ on } \mathcal{U}_{min}(w) \text{ on } w <: v \text{ on } w$. And from Proposition 15, we get $\Delta(u,w) \text{ on } \mathcal{U}_{min}(w) <: v$. The second part of the proof is symmetrical.* □

Let us finally define the simplification relation $\rightsquigarrow$ between clock constraints. Its definition is given in Figure 4.8. Any new variable appearing in right-hand side of the simplification relation is assumed to be fresh.

Theorem 12 (preservation of satisfiability) *If S is satisfiable and $S \rightsquigarrow S'$ then S' is satisfiable.*

Proof. *Proposition 12 authorizes to sample (to slow down) the system and will be used throughout the proof. Let us consider every relation in Figure 4.8.*

(SUP), (INF) and (CYCLE). *Presevation of satisfiability is a direct application of Propositions 2 and 5.*

(EQUAL). *This rule preserves satisfiability: it just subsamples a pair of variables.*

(CUT). *By definition of $\mathcal{U}_{min}$ and $\mathcal{U}_{max}$, the right-hand side of the relation is a sufficient condition of satisfiability.*

Conversely, consider a solution $\alpha_1 = \alpha \text{ on } v_1$ and $\alpha_2 = \alpha \text{ on } v_2$. Let $V_1 = \mathcal{V}(v_1, \mathcal{U}_{min}(w))$ and $V_1' = \mathcal{V}'(v_1, \mathcal{U}_{min}(w))$, and replace α by α' on V_1. We have $\alpha_1 = \alpha' \text{ on } V_1' \text{ on } \mathcal{U}_{min}(w)$. Let us choose $\alpha_3 = \alpha' \text{ on } \Delta(V_1', w)$; From Proposition 16, we have $\alpha_1 <: \alpha' \text{ on } V_1' \text{ on } \mathcal{U}_{max}(w) <: \alpha_3 \text{ on } \mathcal{U}_{max}(w)$.

We also have $V_1' \text{ on } \mathcal{U}_{min}(w) \text{ on } w <: V_1 \text{ on } v_2 \text{ on } w$, hence Proposition 16 yields $\Delta(V_1', w) \text{ on } \mathcal{U}_{min}(w) <: V_1 \text{ on } v_2$. Since $\alpha_2 = \alpha' \text{ on } V_1 \text{ on } v_2$, we have $\alpha_3 \text{ on } \mathcal{U}_{min}(w) <: \alpha_2$. The right-hand side of the relation is thus satisfiable.

(FORK) and (JOIN). *The proof is very similar: choosing the same α_3 satisfies both inequalities on α_1 and α_2 simultaneously.*

(SUBST). *Consider the form of the inequality I on α. The right-hand side of the relation is of course a sufficient condition of satisfiability. It is also clear that it is necessary when the inequality does not belong to a circuit. Assuming it belongs to a circuit, simplify the system through the systematic application of all other rules, enforcing that no inequality belongs to multiple simple circuits. A retiming argument [LS91] shows that, if the system is satisfiable, then there is a solution such that all inequalities in a given circuit but (at most) one are converted to equalities: considering a solution with at least two strict inequalities, split the circuit by renaming the common clock variable, choosing one name for the path from one inequality to the other and another one on the other path, unify any one of the broken inequalities to effectively remove this inequality from the solution.*

The proof is symmetical for the second form of I.

□

Rule (EQUAL) is only provided to factor the unification step out of the (CUT), (FORK) and (JOIN) rules. As a consequence, in the following resolution algorithm, we assume rule (EQUAL) is an enabling simplification, applied once before each rule (CUT), (FORK) and (JOIN).

Theorem 13 (resolution algorithm) *The set of rules in Figure 4.8 defines a non-deterministic, but always terminating resolution algorithm:*

1. *the tree of simplifications $S \rightsquigarrow S'$ is finite;*

2. *if S is satisfiable, there is a sequence of rule applications leading to the empty set.*

Proof. *The proof is based on the graph structure induced by S.*

(SUP) and (INF) strictly reduces the number of acyclic paths. (EQUAL) is only used once for each application of (CUT), (FORK) and (JOIN). The $w_1 \neq w_2$ condition guarantees it can only be applied a finite number of times. A systematic application of (SUP), (INF), (CUT), (FORK) and (JOIN) leads to a system where no inequality belongs to multiple simple circuits. This enables (SUBST), which strictly reduces the length of any circuit or multi-path subgraphs. (CYCLE) reduces short-circuits on a single variable.

Any ordering in the application of these rules terminates, and yields the empty set when S is satisfiable. □

As a corollary:

Theorem 14 (completeness) *For any expression e, and for any period and clock environments P and H, if e has an admissible clock type in P,H for the relaxed clock calculus, then the type inference algorithm computes a clock ct verifying $P,H \vdash_s e : ct$.*

Intuitively, if the type constraints imposed by the clock calculus are satisfiable, then our resolution algorithm discovers one solution. This strong result guarantees the clock calculus's ability to accept all programs with periodic clocks that can be translated to a strictly $(0-)$synchronous framework.

Completeness would be easier to derive from principality, i.e., from the existence of a most general type for every expression [Pie02, AW93]. Yet the unification of clock stream types is not purely structural (it exploits the properties of the `on` operator), and there are many ways to solve an equation on clock types. There is not much hope either that the system of clock constraints can be solved by a set of confluent rules, since multiple solutions are often equivalent up to retiming [LS91].

Finally, although Theorem 13 proves completeness, our resolution algorithm does not guarantee anything about the quality of the result (total buffer size, period length, rate of the common clock).

4.4 Translation Procedure

When a network is associated with a system of clock inequalities where not all of them are simplified into equalities, its execution is undefined with respect to the semantics of 0-synchronous programs. Buffer processes are needed to synchronize producers with consumers.

4.4.1 Translation Semantics

Consider the input clock ck `on` w and the output period ck `on` w', with $w \preceq w'$. To fully synchronize the communication, we insert a new buffer node $\mathtt{buffer}_{w,w'}$ with clock $\forall\beta.\beta\ \mathtt{on}\ w \rightarrow \beta\ \mathtt{on}\ w'$; w (resp. w') states when a *push* (resp. *pop*) occurs.

Proposition 17 (buffer size) *Consider two synchronizable infinite binary words w and w' such that $w \preceq w'$. The minimal buffer to allow communication from w to w' is of size*

$$size(w,w') = \max(\max_{p,q}(\{q-p \mid [w']_p \geq [w]_q\}),0).$$

Communication from w to w' is called $size(w,w')$-synchronous.

On periodic words, this size is effectively computable thanks to Remark 1.

Proof. *This is the maximal number of pending writes appearing before their matching reads, hence a lower bound on the minimal size. It is also the minimal size, since it is possible to implement a size n buffer with n registers.* □

For the simplified downscaler, buffer size is equal to 1, since clock $0^{9600}(1000010000000\,10000000100)$ may take at most one advance tick with respect to clock $0^{9603}(100000)$. For the real downscaler, we automatically computed the size **880**. (This is not the transposition buffer in the reorder node: its size is defined in the specification and not inferred from the clocks.)

Let us now define a *translation semantics* for programs accepted with the relaxed clock calculus. This will enable us to state the cornerstone result of this work, namely that programs accepted with the relaxed clock calculus can be turned into synchronous programs which are accepted by the original clock calculus. This is obtained through a program transformation which inserts a buffer every time a strict inequality on stream clock types remains after resolution. Because a buffer is itself a synchronous program, the resulting translated program can be clocked with the initial system and can thus be synchronously evaluated. This translation is obtained by asserting judgment $P,H \vdash_s e : ct \Rightarrow e'$, meaning that in the period environment P and the clock environment H, the expression e with clock ct is translated into e'. The insertion rule is:

$$\text{(TRANSLATION)}\ \frac{P,H \vdash_s e : ck\ \mathtt{on}\ w \Rightarrow e' \quad w <: w'}{P,H \vdash_s e : ck\ \mathtt{on}\ w' \Rightarrow \mathtt{buffer}_{w,w'}(e')}$$

Other rules are simple morphisms.[8]

4.4.2 Practical Buffer Implementation

From the definition in Section 4.3.3, one may define a custom buffer process with the exact clock type to resynchronize a communication. Yet this definition suffers from the intrinsic combinatorics of $(0-)$synchronous communication between periodic clocks (with statically known periodic clocks). We propose an alternative construction where the presence or absence of data is captured by dynamically computed clocks. The memory and code size

[8] Notice the (CTR) rule shifts a clock constraint imposed by the programmer; this rule will often lead to the insertion of a synchronization buffer, triggering the (TRANSLATION) rule indirectly.

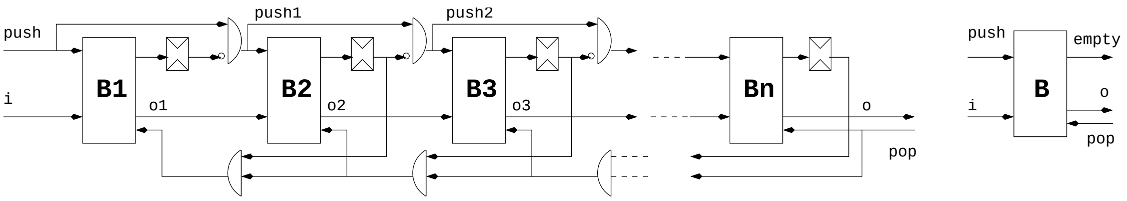


Figure 4.9: A synchronous buffer

become linear in the buffer size, which is appropriate for a practical implementation. The downside is that static properties about the process become much harder to exhibit for automated tools (model checking, abstract interpretation): in particular, it is hard to prove that the code actually behaves as a FIFO buffer when at most *n* tokens are sent and not yet received.

```
let node buffer1 (push, pop, i) = (empty, o) where
  o = if pempty then i else pmemo
  and memo = if push then i else pmemo
  and pmemo = 0 fby memo
  and empty =
    if push then if pop then pempty else false
    else if pop then true
         else pempty
  and pempty = true fby empty
```

Figure 4.10: Synchronous buffer implementation

A buffer of size one, called 1-buffer, can be written as a synchronous program with three inputs and two outputs. It has two boolean inputs `push` and `pop` and a data `i`. `o` and `empty` are the outputs. Its behavior is the following: the output `o` equal `i` when its internal memory was empty and equals the internal memory otherwise. Then, the memory is set to `i` when `push` is true. Finally, the `empty` flag gives the status of the internal memory. If a `push` and a `pop` occur and the memory is empty, then the buffer is bypassed. If a push occurs only, `empty` becomes false. Conversely, if a `pop` occurs then the memory is emptied. This behavior can be programmed in a synchronous language. Figure 4.10 gives an implementation of this buffer in a strictly synchronous language.[9] Buffers of size *n* can be constructed by connecting a sequence of 1-buffers as shown in Figure 4.9. To complete these figures, notice the boolean streams `push` and `pop` need to be computed explicitely from the periodic words w and w' of the output and input stream clocks.

Finally, because safety is already guaranteed by the calculus on periodic clocks, a synchronous implementation for the buffer is not absolutely required. An array in random-access memory with head and tail pointers would be correct by construction, as soon as it satisfies the size requirements.

4.4.3 Correctness

We define judgment $P,H \vdash e : ct$ to denote that expression e has clock ct in the period environment P and the clock environment H, for the *original* 0-synchronous system. The following result states that any program accepted by the relaxed clock calculus translates to an equivalent 0-synchronous program (in terms of data-flow on streams). This equivalent program has the same clock types.

Theorem 15 (correctness) *For any period environment P and clock environment H, if $P,H \vdash_s e : ct \Rightarrow e'$ then $P,H \vdash e' : ct$.*

The proof derives from the subtyping rule underlying $\vdash_s$ judgments: classical subtyping theory [Pie02, AW93, Pot96] reduces global correctness to the proof of local 0-synchronism of each process composition in the translated

[9] LUCID SYNCHRONE [CP03]; distribution and reference manual available at `www.lri.fr/~pouzet/lucid-synchrone`.

program (including `at` clock constraints). This is guaranteed by the previous buffer insertion scheme, since each buffer's signature is tailored to the resynchronization of a pair of different but synchronizable clocks. This ensures the translated program is synchronous.

4.5 Synchrony and Asynchrony

A system that does not have a single synchronous clock is not necessarily asynchronous: numerous studies have tackled with relaxed or multi-clocked synchrony at the hardware or software levels. We only discuss the most closely related sudies, a wide and historical perspective can be found in [Cas01].

There are a number of approaches to the specification and design of hybrid hardware/software systems. Most of them are graphical tools based on process networks. Kahn process networks (KPN) [Kah74] is a fundamental one, but it models only functional properties, as opposed to structural properties. KPN are used in a number of tools such as YAPI [dKES$^+$00] or the COSY project [BKK$^+$00]; such tools still require expertise from different domains and there is no universal language that combines functional and structural features in a single framework.

Steps towards the synchronous control of asynchronous systems are also conducted in the domain of synchronous programming languages, such as the work of Le Guernic et al. [LTL03] on Polychrony. This work targets the automatic and correct by construction refinement of programs, in the same spirit as our clock composition, but it does not consider quantitative properties of clocks. StreamIt [TKA02] is a language for high performance streaming computations that tackles mainly stream-level and algebraic optimization issues.

Ptolemy [BHLM94] is a rich platform with simulation and analysis tools for the design of embedded streaming systems: it is based on the synchronous data-flow (SDF) model of computation [EAL87]. Unlike synchronous languages, SDF graphs cannot express (bounded or not) recursion and arbitrary aperiodic execution. They are not explicitly clocked either: synchrony is a consequence of local balance equations on periodic execution schemes. The SDF model allows static scheduling and is convenient for the automatic derivation of timing properties [MBvM04], but the lack of clocks weakens its amenability for formal reasoning and correct-by-construction generation of synchronous code, with respect to synchronous languages [HCRP91, BCE$^+$03]. Interestingly, *n*-synchronizable clocks seem to fill this hole, leading to the definition of a formal semantics for SDF while exposing the precise static schedule to the programmer (for increased control on buffer management and code generation). Further analyses of the correspondence between the two models are left for future work.

4.6 Conclusion

We proposed a synchronous programming language to implement correct-by-construction, high-performance streaming applications. Our model addresses the automatic synthesis of communications between processes that are not strictly synchronous. In this model, we show that latencies and buffer requirements can be inferred automatically. We extend a core data-flow language with a notion of periodic clocks and with a relaxed clock calculus to compose synchronous processes. This relaxed synchronous model defines a formal semantics for synchronous data-flow graphs, building a long awaited bridge with synchronous languages. The clock calculus and the translation procedure from relaxed synchronous to strictly synchronous programs are proven correct, and the associated type inference is proven complete. An implementation in the synchronous language LUCID SYNCHRONE is under way and was applied to a classical video downscaler example. We believe this work widens the scope of synchronous programming beyond safety-critical reactive systems and circuit synthesis, promising increased safety and productivity in the design and optimization of a large spectrum of applications.

Chapter 5

Perspectives

To conclude this manuscript, we build on our results and understanding to propose a structured analysis of the immediate and longer-term perspectives.

Our work-plan intensionally extends the recent results presented in this thesis. Although this may look like a rather closed point of view (w.r.t. the global research area), this plan contributes to a tighter integration of the research conducted so far. The aim is to prioritize the achievement of key contributions involving the deepest level of specialization, while maximizing the impact on applied domains. The proposed work is also wide enough to cover several challenges identified by the research community, and to drive the activity of multiple research groups. As a matter of fact, most of these perspectives are supported by research projects involving the ALCHEMY group (from European and French funding agencies) and collaborations with the best research group from academia and industry, in Europe and abroad.

We hope this manuscript will help other research groups share, criticize and confront our analysis, and thus contribute to the achievement of significant advances in high-performance computing.

This chapter addresses the interplay of compiler foundations and technology with

1. programming languages,
2. program generators,
3. processor architectures,
4. runtime systems,
5. and the associated tool (infrastructure) development efforts.

5.1 Compilers and Programming Languages

We believe the future of scalable and efficient computing systems lies in parallel programming languages with strong semantical guarantees, allowing to maximize productivity, robustness and portability of performance. This section is a first attempt at shaping an "ideal" parallel programming model. The goal is to let the programmer express *most* of the parallelism, in a *deterministic*, *compositional* (or modular) and *compiler-friendly* (or overhead-free) semantics.

5.1.1 First step: Sparsely Irregular Applications

Starting from the experience of the data-flow language StreamIt, we believe we can dramatically improve expressiveness, parallelism extraction, optimization and mapping opportunities on specific hardware. This work will first be targeted towards regular data and computationally intensive applications, with sparsely irregular control (mostly mode transitions).

Most of the parallelism. Most parallel languages ask the programmer to make sensible choices in the exposition of coarse grain parallelism. Yet to allow multi-level parallelism to be efficiently exploited on a variety of targets, it is absolutely necessary to expose the fine-grain data-flow structure of the computation. To simplify the compilation and runtime, we first assume a restricted form of control-flow, operating at a sparse, higher "reconfiguration" level.

In a data-flow execution model, there is no need for an explicit distinction between data and control parallelism. Processes are functions with private memory/state; they can be composed and replicated. Communication and control is explicit, and can be provided with an additional interface for comfortable data parallel programming and for the flexible expression of functional pipelines. This interface may take the form of semantically rich communication and partially-parallel computation skeletons, built on top of the abovementioned data-flow primitives, like reductions/scans, collective communications. This interface should also provide a clean atomic execution and synchronization semantics at the higher level of control.

Deterministic. To preserve the Kahn principle [Kah74] (determinism and absence of schedule or resource-induced dead-locks), there must be no information flow between processes without explicit communications, assuming non-blocking writes and blocking reads on FIFO channels. Non-FIFO communications can be seen as "peek/poke" operations into implicitly unbounded single-assignment buffers (but practically bounded through static analysis), which can in turn be translated into FIFO operations [TKA02], but compiled much more efficiently with circular buffers, address generators or rotating register files. Communications can be hierarchically structured to reflect the organization of data or computation itself: one write/read operation in a high-level process may be further detailed into a deterministic sequence of finer-grain communications in nested processes. The mapping of these semantically abstract communications onto target-specific primitives is completely hidden.

Compositional. Determinism offers composition (or modularity) for free, as long as liveness and equity properties are satisfied (prerequisites of the Kahn principle). Equity depends on the compilation flow, for the larger part, in a statically controlled architecture. Liveness must be proven at compilation time by static analysis (e.g., checking for delays in circuits of the data-flow graph).

Preserving compositionality of resource, placement and schedule properties is much more difficult, but synchronous Kahn networks [CP96, CGHP04] are a great progress in this direction, as discussed in the next paragraph.

Compiler-friendly. To ensure early satisfaction of the resource and real-time constraints in the compilation flow, the data-flow semantics must be complemented with implicit and local scheduling information to reason globally about the application-architecture mapping. This is the contribution of the synchronous execution model to high-performance parallel programming; constraining all computations to take place at well defined instants in a global, totally ordered, logical time (hence the term "synchronous") allows for much more advanced compilation strategies and optimizations. The lack of such semantical information is a major reason for StreamIt's inability to harness multi-level, heterogeneous and fine-grain parallelism.

A synchronous distributed system does not have any hidden communication buffer: all communications are "logical wires" and do not need any particular storage beyond the duration of the "instantaneous clock tick". In practice, depending on the compilation strategy, registers, memory or communication elements may be required. We advocate for a pragmatic implementation of this principle, to go much beyond the limitations of StreamIt [TKA02] which are those of the SDF [EAL87]. We propose to use two levels of clocks:

- asymptotically periodic, or piecewise affine ones to model the start-up and steady state of the regular data-flow level of the computation;

- arbitrary boolean clocks (which depend on external signal or internal reactions) to model the reconfigurations and trigger transient executions of the application, effectively modeled by an bounded aperiodic but statically known behavior of the application.

The periodic/affine level of clocks yields a statically manageable hierarchy of computations that may be converted into compact, efficient loop nests, amenable to further optimizations and fine-grain parallelization. These clocks also allow to relax the synchronous hypothesis by allowing to compose "almost synchronous" processes and infer the intermediate buffers automatically to statically allocate memory/communication resources [CDE$^+$06]. Finally, these logical clocks are needed to generate efficient code and manage resources statically, but they can be overridden in the optimization flow, through multi-dimensional retiming techniques.

The higher level of more expressive but semantically poorer clocks requires dynamic control in the generated code (nested `if` conditionals), and resembles the compilation of most synchronous languages [BCE$^+$03].

Sparsely irregular. The abovementioned two-level clocks allow to model irregular reconfigurations or "mode transitions". Typically, all modes and all transitions would be statically defined in the application, letting the compiler best optimize the schedule and resource mapping for each transient and steady state. Early feedback on the satisfaction of the resource and real-time constraints can be gathered, facilitating the adaptation of the partitioning in a manual or automatic design-space exploration of the application-architecture mapping.

On the data and communication side, the non-FIFO communications can be translated into bounded arrays. Specific treatment of affine subscripts is very important, but allowing indirect subscripts is important for some rendering algorithms and for mesh computations.

5.1.2 Second step: General-Purpose Parallel Clocked Programming

To broaden the acceptation of data-flow concurrency in parallel computing (beyond streaming applications and systems), we revisit the basic principles of synchronous data-flow computing with a more compiler-centric point of view. For example, instead of emphasizing on the synchrony of the “rendez-vous” between independent threads and the instantaneity of the local computations, we prefer to develop the ability of the concurrency model to compute directly on clocks as first-class citizens. This point of view does not match all flavors of synchrony, in particular it radically departs from the Esterel point of view where clocks should not exist in the program semantics itself. There are numerous reasons why clocks — and the associated semantical restrictions and clock-directed compilation strategies — are necessary to enable “efficient-by-design” compilation methods (e.g., generating sequential code with nested loops and no conditional control-flow from multiple processes with different clocks).

The extension to more general, less regular computations, is also a longer term challenge. Our current approach is to investigate how dynamic parallelization and adaptation techniques can be combined with the data-flow programming model, to allow for limited non-deterministic, speculative or data-dependent computations to be efficiently compiled on a parallel architecture. The section on runtime systems below provides more insights about this, and we advocate for a tight cooperation between deterministic and non-deterministic forms of parallelism — atomic transactions, with inspector-executor instrumentation or speculation — with a core data-flow concurrency model.

5.2 Compilers and Program Generators

The quality of compiler-optimized code for high-performance applications is far behind what optimization and domain experts can achieve by hand. Although it may seem surprising at first glance, the performance gap has been widening over time, due to the tremendous complexity increase in microprocessor and memory architectures, and to the rising level of abstraction of popular programming languages and styles. We wish to further explore in-between solutions, neither fully automatic nor fully manual ways to adapt a computationally intensive application to the target architecture. As hinted to in the introduction, much progress was achieved, not by improving the compiler, but through the design of application-specific program generators, a.k.a. active libraries [VG98]. The most advanced project in this area is SPIRAL [PSX$^+$04].

We are interested in improving language support for domain experts to implement such active libraries. This motivation is apparently shared by language designs in Fortress [ACL$^+$06]. Our early work is based on multi-stage languages. In particular, we used MetaOCaml to experiment with off-line and online generation of high-performance programs [CDG$^+$06]. The main advantages of this approach are the ability to generate code that does not trigger future-stage compilation errors (syntax, structure, type), and the seamless integration of values produced in various execution stages (for online optimization and specialization).

Yet our results have been mitigated by the constraints on code introspection (opening code expressions) associated with the simultaneous enforcement of type safety for all stages. The ability to implement generic and reusable program generator components is also restricted by the need to express some transformations on a higher level, non-syntactic representation, using unconventional domain-specific denotations, polyhedral encodings of iterative computations, etc. Some progresses can be expected from extended type systems to manage the recapture of variable names exposed through the opening of code expressions.

Another interesting perspective is the extension of safety guarantees to abstract properties like memory dependences. This would allow semantics-preserving transformations to take place on well-formed code expressions or denotations. Our first proposal, the X-language, goes in this direction. It combines multi-stage programming with programmable semantics-preserving loop transformation primitives and search-space traversal capabilities [DBR$^+$05].

Eventually, in collaboration with Denis Barthou, we are investigating the complementation of program generators with a resolution-based search engine. Using Prolog or Stratego [Vis01] rules, it is quite natural to design an iterative optimization system based on term-rewriting, guided by profile feedback and performance models. This allows direct search techniques (optimizations based on direct performance measurements) to be accelerated and focused in a narrower space thanks to the knowledge of a domain expert. This knowledge is implemented as optimization *goals* of the rewriting system. Therefore, we call this approach *goal-driven iterative optimization*. A major challenge consists in inverting the performance models to infer optimization sub-goals from partial goal fulfillments and partial transformation sequences.

5.3 Compilers and Architectures

The biggest challenge for the design of future architectures is to achieve scalability of performance. Yet, scalability is not so useful if efficiently and programmability are neglected. It is well known from VLIW processor design that the main improvements in efficiency come with progresses in compiler technology. To make an architecture resource-efficient, the balance may vary between functional units, memory and communication; however, the invariant goal is to minimize the size of control structures in favor of smarter software support (at runtime and compile-time).

We believe that future processors will necessarily follow a distributed memory model and feature a heterogeneous, coarse-grain reconfigurable mesh of resources. Unlike FPGAs and fine-grain reconfigurable devices (which waste too many transistors in hardware lookup tables), coarser grain reconfigurable processors with heterogeneous and distributed resources offer the best tradeoff in terms of scalability and efficiency. The RAW machine was a pionneer in this direction, but major challenges remain to adapt the computing paradigm to the capabilities of code generators and optimization techniques, or alternatively, to involve the programmer in the mapping process while preserving productivity.

5.3.1 Decoupled Control, Address and Data Flow

Our approach to the efficiency challenge takes two consecutive steps.

The first step focuses on embedded streaming applications and processors, with regular computation and communication patterns. Inspired by the early DSP designs (Harvard architecture), it makes sense to strictly decouple the architecture into control, address generation and data-flow streams. Our previous work in the SANDRA project [CDC$^+$03], in collaboration with Philips Research, was a first attempt to design a compiler for such a decoupled architecture for stream-processing. Improvements in polyhedral code generation and automatic distribution will be very valuable for regular applications. Yet more general and irregular processing will benefit from a explicitly parallel data-flow programs with synchronous clocks, to guide the mapping and scheduling on a distributed decoupled architecture. In this direction, we are working on explicitly modeling hardware resources and the mapping of operations and communications in the application to those resources, as the composition of synchronous processes. The design of a common synchronous language interface between the programmer and a suite of optimization tools would be a fundamental progress. It would also be of immediate use throughout the design of a massively parallel real-time embedded system, as an exchange representation between signal-processing experts, application programmers, optimization and architecture experts.

The second step will also be language-centric, and will address general-purpose computing. Building on the clock abstraction of the synchronous data-flow paradigm, we know it is possible to design a concurrent programming language and to extend compilation techniques designed in the first step, to target more general clock and process semantics, possibly with a graceful degradation in performance as irregularity grows. Neither decoupled architectures nor synchronous languages are intrinsicly domain-specific: we believe the current state-of-the-art, inertia and misuse of vocabulary[1] make them appear so.

5.3.2 Fine-Grain Scheduling and Mapping

Modulo scheduling is the dominant family of heuristics for software pipelining. Its success lies in its scalability and its ability to produce compact code with near-optimal initiation intervals. This success comes with a heavy bias towards asymptotic performance for high trip-count loops, leaving considerations about single-iteration make-span and register usage as second-class optimization criteria. Empirical results also indicate that modulo-scheduling heuristics achieve low complexity at the expense of chaotic performance results (on realistic resource models). Eventually, modulo scheduling lacks an integrated way to reason about code size (unrolling factor) and to model

[1] Synchronous languages do not require synchronous execution and can model arbitrary non-reactive semantics [CP96, CGHP04].

resource constraints that are not transparent to the data dependence graph (register spill and reload, clustered register files, instruction selection). The combination of this overall success and intrinsic limitations has driven numerous theoretical and practical studies, but with a relatively low impact on actual compiler designs at the end of the road.

We are working on a new cyclic scheduling algorithm, called Pursuit Scheduling. It is a bi-criteria heuristic for software pipelining, based on a Petri net representation of dependence and resource constraints. The aim is to simulate modulo scheduling, but extend it into a bi-criteria optimization problem, to improve both the initiation interval and make-span. We attempt to preserve the best of modulo scheduling heuristics, while improving performance stability and allowing for a variety of optimization goals and resource constraints. Unlike existing Petri net scheduling approaches, our algorithm does not iterate until a fix point to ultimately recognize a periodic execution kernel. Instead, it complements the traditional Petri net semantics with an original transition-firing rule that preserves the distinctive code-size control of modulo scheduling. Our current results are encouraging: when directing the optimization towards asymptotic performance, our experimental results match or outperform state-of-the-art modulo scheduling heuristics. Our results also show benefits when optimizing loops with low trip-counts, which are common in multimedia routines and hierarchically tiled high-performance libraries.

Given the flexibility of the model, it is very attractive to extend this heuristic to multidimensional software pipelining. Indeed, when scheduling quasi-periodic data-flow processes on a massively parallel chip multiprocessor, the optimization problem becomes intrinsicly multi-criteria. The problem consists in optimizing code size, buffer size, computation throughput and computation latency, but none of these dimensions can be neglected, since the figures in each dimension may vary by multiple orders of magnitude (unlike traditional software pipelining problems). We believe that programmer intervention will be necessary to drive the optimization towards reasonable sub-spaces of this daunting problem. We are currently studying, as a preliminary example, the scheduling of Synchronous Data-Flow graphs (SDF) [EAL87] on the Cell processor. When stepping from regular quasi-periodic applications to general-purpose ones, our previous work on Deep Jam — a generalization of nested unroll-and-jam for irregular codes — will be quite valuable [CCJ05].

5.4 Compilers and Runtime Systems

Embedded systems and large-scale parallel architectures both require dynamic adaptation to make most effective use of computing resources. These research perspectives would not be complete without considering staged compilation, as well as dynamic parallelization and optimization techniques.

5.4.1 Staged Compilation and Learning

In a just-in-time (JIT) compilation environment, optimizations are staged, i.e., split into off-line and runtime program manipulations. The success of this compilation model suggests additional improvements could arise from a tighter coupling between program analyses and transformations at different stages of the compilation/execution. We are particularly interested in progresses in three areas.

1. Enabling new, more aggressive optimizations to take place in JIT or runtime compilers, using annotations of the intermediate language (bytecode) to pass static properties or predigested results of operation research algorithms. For example, software pipelining and global scheduling algorithms are usually too expensive beyond classical static compiler environments. This could change provided a sparse and accurate encoding of memory dependences and/or partial schedule hints can be passed to a JIT compiler.

2. Intermediate language annotations can convey messages for future-stage code generation, including code specialization, loop unrolling and procedure inlining. This may have a strong impact on code size and benefit to embedded systems research.

3. We believe in the combination machine learning and phase/context adaptation, passing features through the compilation stages to postpone the key optimization and parallelization decisions to the "right execution/compilation stage". This approach builds on our previous work on runtime iterative optimization [FCOT05].

5.4.2 Dynamically Extracted Parallelism

A variety of computational problems benefit from the dynamic extraction of parallelism. For example, it does not make much sense to extract fully-static parallelism from sparse matrix computations, or from finite element methods on irregular meshes, in general. In some cases, speculation can further improve scalability in extracting parallelism at a coarser grain or in reducing load imbalance.

The most promising work in this area are those of Lawrence Rauchwerger, related work by Josep Torrellas, and the recent hype on transactional memory in computer architecture.

- Hybrid analysis performs a symbolic static analysis to synthesize the minimal dynamic (in)dependence test to parallelize a loop [RRH03]. Extending this to more general control and data structures may bring significant improvements in irregular, computationally intensive applications, with very little overhead. This approach is compatible with most embedded system constraints. Similarly, STAPL proposes a library-centric, container-centric parallel programming model based on the C++ STL [AJR$^+$01]. Besides its C++ specificities, this model combines dynamic and static information to exploit vast amounts of parallelism on distributed architectures. Combining both approaches, allowing the compiler to reason about (in)dependence in general-purpose computations, seems the most promising approach. Once again, this research area needs library, program generators and compilers to be tightly integrated.

- Speculative parallelization can be done on software or hardware [OHL99, RP99, CMT00, LTC$^+$06]. It has an interesting potential in speeding up sequential and parallel applications. Yet its intrinsic overheads (memory management, communications, squash and commit, useless computations) limits its practical applicability. Transactional memory models (hardware or software based) [MCC$^+$06, HMJH05] and programming languages [ACL$^+$06] are an interesting progress over lock-based multi-threaded programs and over the fully automated extraction of speculative threads. We are interested in minimizing the impact on hardware resources (using speculation when strictly necessary, and reducing on-chip storage for atomic transaction state), and in the combination of transactional semantics with deterministic, synchronous data-flow semantics.

Regarding the potential of transactional memory semantics, let us quote Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy [HMJH05] (Section 2, page 2) describing a software transactional memory system built on Concurrent Haskell:

> Perhaps the most fundamental objection [...] is that lock-based programs do not compose: correct fragments may fail when combined. For example, consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item `A` from table `t1`, and insert it into table `t2`; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement. [...] In short, operations that are individually correct (insert, delete) cannot be composed into larger correct operations.

The ability to compose parallel programs into correct larger ones is nothing new. The already mentioned Kahn property of data-flow systems already offers this benefit, in a distributed but more constrained — deterministic — concurrency model. We encourage further comparisons in terms of scalability, efficiency and expressiveness, aiming for the integration of the two models.

Eventually, given the complexity and intrication of decisions associated with deferring analyses to runtime or with speculation, it is obvious that this research will benefit from advances in machine learning compilation [ABC$^+$06].

5.5 Tools

This chapter closes with a discussion of tool and development issues. Perspectives in this area are not secondary. Compilation research has long suffered from the lack of interoperability between research prototypes, production compilers and academic language front-ends. Given the high complexity of any modern compiler and programming language, we defend the idea that academic research will suffer from ever-increasing inefficiencies if not embracing a widespread production compilation framework.

Practically, this means choosing GCC, since all other platforms are either very narrow in terms of retargetability and language support, or not robust and maintainable enough in the long term. We are not encouraging *all* developments to take place in GCC, only those which have the ambition to *survive the research project* that initially motivated them.

The GCC is a multi-language, multi-target, and multi-OS cross-compiler with about 2.5 million lines of (mostly C) code. The development started in 1984 as part of the GNU project of the Free Software Foundation (FSF). In 2005, GCC 4.0 was released thanks to the efforts of many developers. It introduces a new, innovative middle-end based on a state-of-the-art SSA-form intermediate representation. This was the result of many years of commitment by major hardware, software and service companies, including Red Hat (Cygnus), Novell (SuSE), IBM, Apple, Intel, AMD, HP, Sony, Code Sourcery, among others.

Although GCC has always been a reference in terms of robustness and conformance to standards, the performance of the compiled code was lagging behind target-specific compilers developed by hardware vendors. The rise of GCC 4.0 eventually allowed to implement modern (static and dynamic) analyses and optimizations, quickly bridging the gap with the best compilers on the market. These improvements also affect embedded and special-purpose processors: e.g., GCC recently achieved the top of the ranking published by the EEMBC telecom benchmark on the PowerPC 970FX processor, with 2.8× speed-up over the previous compiler.

GCC 4.2 features more than 170 compilation passes, two thirds of them playing a direct role in program optimization. These passes are selected, scheduled, and parameterized through a versatile pass manager. The main families of passes can be classified as:

- interprocedural analyses and optimizations;
- profile-directed optimization (interprocedural and intraprocedural);
- induction variable analysis, canonicalization and strength-reduction;
- loop optimizations;
- automatic vectorization;
- data layout optimization.

More advanced developments are in progress. We identified three major ones with a direct impact on high-performance embedded systems research:

- link-time optimization (towards just-in-time and dynamic compilation), with emphasis on scalability to whole-program optimization and compatibility with production usage;
- automatic parallelization, featuring full OpenMP 2.5 support and evolving towards automatic extraction of loop and functional parallelism, with ongoing research on speculative forms of parallelism.

Outside of GCC, the two traditional alternatives would be:

1. The source-to-source model, by far the most popular in research environments, during the last two decades. It is much less attractive now, for two reasons. Experiences with the new infrastructure of GCC show comparable development time and no significant increase in complexity of the intermediate representation or programmer interface: in practice, the design freedom benefits do not offset the development overheads, except on limited toy prototypes. Second, the lack of an integrated compilation flow, broken by one ore more conversions to a programming language, is a source of runtime overheads (not speaking about compile-time overheads of course), semantical mismatches and design bugs, usage restrictions.[2]

   We believe the only and most significant advantage of source-to-source compilation is portability, not ease of development. Given the openness of GCC (license-wise, language-wise and target-wise), this advantage is not very significant. Of course, some research fields still require the use of proprietary back-ends, including hardware synthesis, and generating code for awkward, but more and more attractive (performance-wise) fine-grained controlled architectures. In the latter case, we encourage researchers to mutualize efforts to bypass these tools and obsolete them through combined applied research and software engineering with industrial third parties.

[2] Notice many compilers use persistent (sometimes semantically well defined) intermediate languages. This is quite different from source-to-source compilation, since arbitrarily rich semantics (including the result of static and dynamic analysis) can be embedded in such intermediate languages.

2. Development in a native compiler prototype. This option has always been less attractive, often due to the closeness of the distribution model or the limitations in retargetability. Since GCC is gradually eating-up the market for most proprietary compilers, this approach does not seem to make sense, except in the domains where GCC is still lagging behind in performance (VLIW and IA64 targets).

   For low-level program manipulation, targeting, e.g., VLIW processors, a lot of work may also take place in external restructuring tools à la SALTO [BRS96]. Combining such approaches with GCC would be highly beneficial to both industrial and academic research.

One may notice a recent surge of top-quality publications on GCC-based advanced compilation research: automatic vectorization in ACM PLDI'06 (Nuzman et al.) and ACM CGO'06 (Nuzman et al.), thread-level speculation in ACM PPoPP'06 (Liu et al.) and ACM ICS'05 (Renau et al.), induction variable recognition in HiPEAC'05 (Pop et al.), statistical analysis for iterative optimization in ACM ICS'05 (Haneda et al.), template meta-programming with concepts in ACM PoPL'06 (Dos Reis et al.). *These articles confirm the worldwide interest for GCC as a mature, competitive research platform*. This is good news for the scientific methodology of our research domain, since the free licensing scheme of GCC encourages not only the publication of the scientific results, but also improves the robustness of the scientific methodology itself, facilitating the reproduction of experiments on real-world benchmarks.

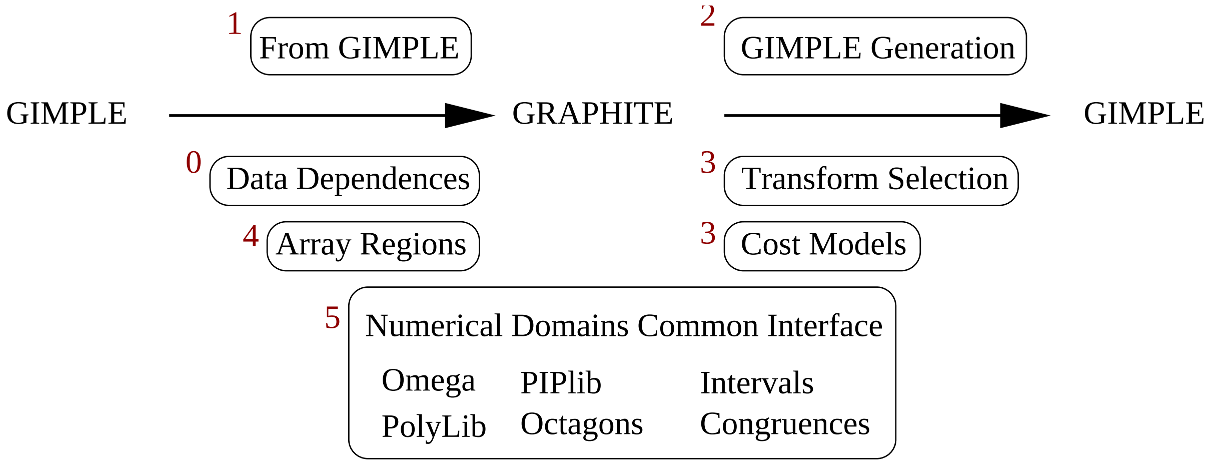


Figure 5.1: Overview of GRAPHITE

Beyond coordination and animation activities within the HiPEAC network, our main efforts have concentrated on induction variable recognition, loop transformations and polyhedral static analysis. Our approach to induction variable analysis complements state-of-the-art classification methods and closed form abstractions [GSW95, LLC96, vE01], and addresses the more fundamental aspect of retrieving precise information from intricate control and data-flow graphs [PCS05]; our algorithm fits particularly well with the normalization and simplification approach of GCC's GIMPLE representation, can be expressed as a handful of Prolog rules. Our approach is compatible with algorithms making use of induction information without explicitely recognizing idioms [WCPH01, RZR04]. Our work also led to the first construction of a formal semantics and the associated conversion algorithm for an SSA language [Pop06]. This result may eventually rejoin the instancewise program manipulation paradigm, as the SSA's formal semantics for a simple `while` language builds directly on instancewise control-point naming. This exciting perspective is also a great source of satisfaction, as this body of work emerged from the most practical research we have ever conducted.[3] This work contributes to the classical loop optimizer of GCC, including loop vectorization, strength reduction, value-range propagation, induction variable canonicalization, and various dependence-based loop transformations. Based on the same work, we are porting our WRaP-IT/URUK polyhedral analysis and transformation tool, from the Open64 compiler to GCC. This plan, called GRAPHITE (GIMPLE Represented as Polyhedra with Interchangeable Envelopes), builds on multiple open libraries developed at INRIA, Paris-Sud University, University of Strasbourg, and École Nationale Supérieure des Mines de Paris [PCB+06]. This development effort will lead to further applied and fundamental research in extending the applicability and effectiveness of the polyhedral model.

[3] To be fair, it really emerged from Sebastian Pop's tenacity and vision, with the experience and rigor of Pierre Jouvelot from ENSMP...

The main components of GRAPHITE, the development priorities, and the important dependences between components are depicted in Figure 5.1. Our development plan contains five stages, numbered on the figure: first the translation from GIMPLE to the polyhedral representation, then the translation back to GIMPLE, the development of cost models and the selection of the transform schedule. The interprocedural refinement of the data dependence information based on the array regions is optional, but it is necessary for gathering more precise informations that potentially could enable more transformations, or more precise transform decisions. Finally, the least critical component is the integration of the numerical domains common interface, based on which it will be possible to change the complexity of the algorithms used in the polyhedral analyses.

We plan to progressively integrate within GCC most of the implementation efforts associated with our research projects. Particularly important is the the development of a convincing, widely applicable and distributed implementation of the parallel programming language designs proposed at the beginning of this chapter. This work is intended to be implemented on top of the existing OpenMP, interprocedural optimization and loop transformation frameworks in GCC, with additional support for managing communications, the optimization of these, and the extended type systems associated with logical clocks for mapping and schedule annotations.

This implementation work will not contribute to the compiler mainline before most of the effective research has been completed. The long-term management of GCC branches with highly sophisticated compilation techniques is a problem. We do not hope that GCC will be a magical solution to the dilemma of academic prototypes: nobody wants to maintain them when it would be most useful in terms of dissemination and transfer. However, we will make constant efforts to involve other members of the GCC community in our research, hence maximize the chances of a successful transfer.

Bibliography

[ABC$^+$06] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *4th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2006.

[ACF03] P. Amiranoff, A. Cohen, and P. Feautrier. Instancewise array dependence test for recursive programs. In *Proc. of the 10th Workshop on Compilers for Parallel Computers (CPC'03)*, Amsterdam, NL, January 2003. University of Leiden.

[ACG$^+$04] L. Almagor, K. D. Cooper, A. Grosul, T.J. Harvey, S.W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, 2004.

[ACL$^+$06] E. Allen, D. Chase, V. Luchangco, C. Flood, J.-W. Maessen, S. Ryu, S. Tobin-Hochstadt, and G. L. Steele. The fortress language specification 0.866. Technical report, Sun Microsystems, 2006.

[ACM$^+$98] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-M. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Intl. Symp. on Computer Architecture*, July 1998.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loop. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 39–50, June 1991.

[AJR$^+$01] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In *Languages and Compilers for Parallel Computing (LCPC'01)*, pages 193–208, 2001.

[AK87] J. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AK02] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[Ami04] P. Amiranoff. *An Automata-Theoretic Modelization of Instancewise Program Analysis: Transducers as mappings from Instances to Memory Locations*. PhD thesis, CNAM, Paris, December 2004.

[AMP00] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *ACM Supercomputing'00*, May 2000.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AW93] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[BACD97] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *ACM Intl. Conf. on Supercomputing (ICS'97)*, pages 340–347, 1997.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[Ban92] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston, 1992.

[Bar98] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, Université de Versailles, France, February 1998. `http://www.prism.uvsq.fr/~bad/these.html`.

[Bas03] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubjana, Slovenia, October 2003.

[Bas04] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France, September 2004.

[BBC$^+$99] M. Barreteau, François Bodin, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John R. Gurd, Jan Hoogerbrugge, Ping Hu, William Jalby, Toru Kisuki, Peter M. W. Knijnenburg, Paul van der Mark, Andy Nisbet, Michael F. P. O'Boyle, Erven Rohou, André Seznec, Elena Stöhr, Menno Treffers, and Harry A. G. Wijshoff. Oceans - optimising compilers for embedded applications. In *Euro-Par'99*, pages 1171–1775, August 1999.

[BCBY04] Christian Bell, Wei-Yu Chen, Dan Bonachea, and Katherine Yelick. Evaluating support for global address space languages on the cray X1. In *ACM Intl. Conf. on Supercomputing (ICS'04)*, St-Malo, France, June 2004.

[BCC98] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *25th ACM Symp. on Principles of Programming Languages (PoPL'98)*, pages 98–106, San Diego, California, January 1998.

[BCC00] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. *Intl. J. of Parallel Programming*, 28(3):213–243, June 2000.

[BCE$^+$03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[BCF97] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. of Parallel and Distributed Computing*, 40:210–226, 1997.

[BCG$^+$03] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 23–30, College Station, Texas, October 2003. Springer-Verlag.

[BEF$^+$96] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[Ber79] J. Berstel. *Transductions and Context-Free Languages*. Teubner, Stuttgart, Germany, 1979.

[Ber00] G. Berry. *The Foundations of Esterel*. MIT Press, 2000.

[BF03] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC Intl. Conf. on Compiler Construction*, number 2622 in LNCS, pages 320–335, Warsaw, Poland, april 2003.

[BF04] C. Bastoul and P. Feautrier. More legal transformations for locality. In *Euro-Par'10*, number 3149 in LNCS, pages 272–283, Pisa, August 2004.

[BF05] Cédric Bastoul and Paul Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(1):3–17, March 2005.

[BFF05] J. Thomson B. Franke, M. O'Boyle and G. Fursin. Probabilistic source-level optimisation of embedded systems software. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, 2005.

[BGGT02] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *Intl. J. of Parallel Programming*, 30(2):65–98, 2002.

[BHLM94] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. J. in Computer Simulation*, 4(2):155–182, 1994.

[BKK$^+$98] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[BKK$^+$00] J.-Y. Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, F. Pétrot, L. Pasquier, E. A. de Kock, and W. J. M. Smits. COSY communication IP's. In *37thDesign Automation Conference (DAC'00)*, pages 406–409, Los Angeles, California, June 2000.

[BLJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[Bou92] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. of Functional Programming*, 2(4):407–423, 1992.

[BRS96] F. Bodin, E. Rohou, and A. Seznec. Salto: System for assembly-language transformation and optimization. In *Workshop on Compilers for Parallel Computers (CPC'96)*, December 1996.

[BRZ03] R. Bagnara, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Int. Symp. on Static Analysis (SAS'03)*, LNCS, San Diego, CA, June 2003. Springer-Verlag.

[Cas01] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *EMSOFT'01*, volume 2211 of *LNCS*, Lake Tahoe, October 2001. Springer-Verlag.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4thACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[CC98] A. Cohen and J.-F. Collard. Instancewise reaching definition analysis for recursive programs using context-free transductions. In *Parallel Architectures and Compilation Techniques (PACT'98)*, pages 332–340, Paris, France, October 1998. IEEE Computer Society.

[CCG96] A. Cohen, J.-F. Collard, and M. Griebl. Data-flow analysis of recursive structures. In *Proc. of the 6th Workshop on Compilers for Parallel Computers (CPC'96)*, pages 181–192, Aachen, Germany, December 1996.

[CCJ05] P. Carribault, A. Cohen, and W. Jalby. Deep Jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *Parallel Architectures and Compilation Techniques (PACT'05)*, pages 291–300, St-Louis, Missouri, September 2005. IEEE Computer Society.

[CDC$^+$03] Z. Chamski, M. Duranton, A. Cohen, C. Eisenbeis, P. Feautrier, and D. Genius. *Ambient Intelligence: Impact on Embedded-System Design*, chapter Application Domain-Driven System Design for Pervasive Video Processing, pages 251–270. Kluwer Academic Press, 2003.

[CDE$^+$05] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. Synchronization of periodic clocks. In *ACM Conf. on Embedded Software (EMSOFT'05)*, pages 339–342 (short paper), Jersey City, New York, September 2005.

[CDE$^+$06] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-sychronous Kahn networks. In *33th ACM Symp. on Principles of Programming Languages (PoPL'06)*, pages 180–193, Charleston, South Carolina, January 2006.

[CDG$^+$06] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, September 2006. Special issue on the First MetaOCaml Workshop 2004.

[CDS96] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Hawaii Intl. Conf. on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*. IEEE Computer Society, 1996.

[CFR$^+$91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CGHP04] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *EMSOFT'04*, Pisa, Italy, September 2004.

[CGP$^+$05] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM Intl. Conf. on Supercomputing (ICS'05)*, pages 151–160, Boston, Massachusetts, June 2005.

[CGT04] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par'04*, number 3149 in LNCS, pages 292–303, Pisa, Italy, August 2004. Springer-Verlag.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5thACM Symp. on Principles of Programming Languages*, pages 84–96, January 1978.

[Cha84] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[CHH$^+$93] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.

[Cho04] F. Chow. Maximizing application performance through interprocedural optimization with the pathscale eko compiler suite. `http://www.pathscale.com/whitepapers.html`, August 2004.

[CK01] A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization. In *ACM Intl. Conf. on Supercomputing (ICS'04)*, pages 92–101, June 2001.

[CLR89] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[CMT00] M. H. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ACM/IEEE Intl. Symp. on Computer Architecture (ISCA'00)*, pages 13–24, 2000.

[Coh99] A. Cohen. *Program Analysis and Transformation: from the Polytope Model to Formal Languages*. PhD thesis, Université de Versailles, France, December 1999.

[Col95] J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Intl. J. of Parallel Programming*, 23(2):191–219, April 1995.

[Col02] J.-F. Collard. *Reasoning About Program Transformations*. Springer-Verlag, 2002.

[Cou81] P. Cousot. *Semantic foundations of programs analysis*. Prentice-Hall, 1981.

[Cou96] P. Cousot. Program analysis: The abstract interpretation perspective. *ACM Computing Surveys*, 28A(4es), December 1996.

[CP96] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ICFP '96: Proceedings of the 1stACM SIGPLAN Intl. Conf. on Functional programming*, pages 226–238. ACM Press, 1996.

[CP03] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In *EMSOFT'03*, pages 134–155, Grenoble, France, 2003.

[Cre96] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris (ENSMP), France, December 1996.

[CSS99] K. D. Cooper, P.J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.

[CST02] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

[CTHL03] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *ACM SIGPLAN/SIGSOFT Intl. Conf. Generative Programming and Component Engineering (GPCE'03)*, pages 57–76, 2003.

[CW99] J. B. Crop and D. K. Wilde. Scheduling structured systems. In *EuroPar'99*, LNCS, pages 409–412, Toulouse, France, September 1999. Springer-Verlag.

[CZT$^+$] C. Coarfa, F. Zhao, N. Tallent, J. Mellor-Crummey, and Y. Dotsenko. Open-source compiler technology for source-to-source optimization. `http://www.cs.rice.edu/~johnmc/research.html` (project page).

[DBR$^+$05] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzaran, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing (LCPC'05)*, LNCS, Hawthorne, New York, October 2005. Springer-Verlag. 15 pages.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond *k*-limiting. In *ACM Symp. on Programming Language Design and Implementation (PLDI'94)*, pages 230–241, Orlando, Florida, June 1994.

[DH00] A. Darte and G. Huard. Loop shifting for loop parallelization. *Intl. J. of Parallel Programming*, 28(5):499–534, 2000.

[DHW$^+$97] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. ProfileMe: Hardware support for instruction level profiling on out-of-order processors. In *In Proceedings of the 30thInternational Symposium on Microarchitecture*, NC, December 1997.

[dKES$^+$00] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. Yapi: Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, California, June 2000. ACM Press.

[DP99] L. De Rose and D. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Trans. on Programming Languages and Systems*, 21(2):286–323, 1999.

[DR94] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, 1994.

[DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, Boston, 2000.

[DSV97] Alain Darte, Georges-Andre Silber, and Frederic Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[EAL87] D. G. Messerschmitt E. A. Lee. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–25, 1987.

[ECH$^+$92] D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. F. Levy, M.S. Paterson, and W.P. Thurston. *Word Processing in Groups*. Jones and Bartlett Publishers, Boston, 1992.

[EK99] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FOSSACS'99*, 1999.

[EM65] C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM J. of Research and Development*, pages 45–68, 1965.

[EP00] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *ACM Symp. on Principles of Programming Languages (PoPL'00)*, pages 1–11, 2000.

[EWO04] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *ACM Symp. on Programming Language Design and Implementation (PLDI '04)*, pages 82–93, 2004.

[FCOT05] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 29–46, Barcelona, Spain, November 2005. Springer-Verlag.

[Fea88a] P. Feautrier. Array expansion. In *ACM Intl. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[Fea88b] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[Fea91] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, December 1992. See also Part I, one dimensional time, 21(5):315–348.

[Fea98] P. Feautrier. A parallelization framework for recursive tree programs. In *EuroPar'98*, LNCS, Southampton, UK, September 1998. Springer-Verlag.

[Fea06] P. Feautrier. Scalable and structured scheduling. *To appear at Intl. J. of Parallel Programming*, 28, 2006.

[FGL99] P. Feautrier, M. Griebl, and C. Lengauer. On index set splitting. In *Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, CA, October 1999. IEEE Computer Society.

[FJ98] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of the ICASSP Conf.*, volume 3, pages 1381–1384, 1998.

[FM97] P. Fradet and D. Le Metayer. Shape types. In *ACM Symp. on Principles of Programming Languages (PoPL'97)*, pages 27–39, Paris, France, January 1997.

[FOK02] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *11thLanguages and Compilers for Parallel Computing*, LNCS, Washington DC, July 2002. Springer-Verlag.

[GC95] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EuroPar'95*, volume 966 of *LNCS*, pages 315–326. Springer-Verlag, 1995.

[GH96] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 1–15, St. Petersburg Beach, Florida, January 1996.

[GMCT03] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *Intl. Conf. on Measurement and Modeling of Computer Systems, ACM SIGMETRICS'03*, San Diego, California, June 2003.

[GMS95] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group-based fields. In *Proc. of the Parallel Symbolic Languages and Systems*, October 1995. See also "Design and Implementation of 81/2, a Declarative Data-Parallel Language, RR 1012, Laboratoire de Recherche en Informatique, Université Paris Sud 11, France, 1995".

[GPRN04] K. Goossens, G. Prakash, J. Röver, and A. P. Niranjan. Interconnect and memory organization in SOCs for advanced set-top boxes and TV — evolution, analysis, and trends. In Jari Nurmi, Hannu Tenhunen, Jouni Isoaho, and Axel Jantsch, editors, *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 15, pages 399–423. Kluwer Academic Press, April 2004.

[GQQ$^+$01] A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware design methodology with the Alpha language. In *FDL'01*, Lyon, France, September 2001.

[GSW95] M. P. Gerlek, E. Stoltz, and M. J. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, January 1995.

[GVB$^+$06] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3), 2006. Special issue on Microgrids. 57 pages.

[H$^+$96] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[Har89] W. L. Harrison. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, October 1989.

[HBKM03] K. Heydeman, F. Bodin, P.M.W. Knijnenburg, and L. Morin. UFC: a global trade-off strategy for loop unrolling for VLIW architectures. In *Proc. Compilers for Parallel Computers (CPC)*, pages 59–70, 2003.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HHN92] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 249–260, San Francisco, Calfifornia, June 1992.

[HMJH05] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'05)*, Chicago, Illinois, 2005.

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Intl. Conf. on Supercomputing (ICS'91)*, Cologne, Germany, June 1991.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, August 1974. North Holland, Amsterdam.

[KAP] KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. `http://www.hp.com/techsevers/software/kap.html`.

[Kel96] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[KKGO01] T. Kisuki, P. Knijnenburg, K. Gallivan, and M. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Parallel Architectures and Compilation Techniques (PACT'00)*. IEEE Computer Society, October 2001.

[KKOW00] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.

[KN02] E. Koutsofios and S. North. *Drawing Graphs With* `dot`, February 2002. `http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`.

[KPR95] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symp. on the frontiers of massively parallel computation*, McLean, 1995.

[KS93] N. Klarlund and M. I. Schwartzbach. Graph types. In *ACM Symp. on Principles of Programming Languages (PoPL'93)*, pages 196–205, Charleston, South Carolina, January 1993.

[KS98] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *25thACM Symp. on Principles of Programming Languages*, pages 107–120, San Diego, CA, January 1998.

[LBCO03] C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation*. Number 3016 in LNCS. Springer-Verlag, 2003.

[LF98] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3):649–671, 1998.

[LF05] S. Long and G. Fursin. A heuristic search algorithm based on unified transformation framework. In *7th Intl. Workshop on High Performance Scientific and Engineering Computing (HPSEC-05)*, 2005.

[LGP04] X. Li, M.-J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *ACM Conf. on Code Generation and Optimization (CGO'04)*, San Jose, CA, March 2004.

[LL97] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24thACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[LLC96] S.-M. Liu, R. Lo, and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 228. IEEE Computer Society, 1996.

[LLL01] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 102–112, 2001.

[LO04] S. Long and M. O'Boyle. Adaptive java optimisation using instance-based learning. In *ACM Intl. Conf. on Supercomputing (ICS'04)*, pages 237–246, St-Malo, France, June 2004.

[LP94] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, April 1994.

[LS91] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1), 1991.

[LSC05] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *International Symposium on High Performance Computer Architecture*, 2005.

[LTC$^+$06] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 158–167, New York, New York, 2006.

[LTL03] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design*, April 2003.

[LW97] V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Intl. J. of Parallel Programming*, 25(6), December 1997. `http://icps.u-strasbg.fr/PolyLib`.

[MAL93] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *20thACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, January 1993.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[MBQ02] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proc. AIMSA*, number 2443 in LNCS, pages 41–50, 2002.

[MBvM04] A.J.M. Moonen, M. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In *Proc. of ProRISC, 15th annual Workshop of Circuits, System and Signal Processing*, pages pages 91–99, Veldhoven, The Netherlands, November 2004.

[MCC+06] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ACM/IEEE Intl. Symp. on Computer Architecture (ISCA'06)*, 2006.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[Nai04] D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. `http://www.gccsummit.org/2004`.

[NNH99] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[O'B98] M. O'Boyle. MARS: a distributed memory approach to shared memory compilation. In *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Pittsburgh, May 1998. Springer-Verlag.

[OHL99] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, California, October 1999.

[Oka96] C. Okasaki. Functional data structures. *Advanced Functional Programming*, pages 131–158, 1996.

[OKF00] M. O'Boyle, P. Knijnenburg, and G. Fursin. Feedback assisted iterative compiplation. In *Proc. LCR*, 2000.

[ORC] Open research compiler. `http://ipf-orc.sourceforge.net`.

[Par66] R. J. Parikh. On context-free languages. *J. of the ACM*, 13(4):570–581, 1966.

[PBCV07] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *ACM Conf. on Code Generation and Optimization (CGO'07)*, San Jose, California, March 2007. To appear.

[PCB+06] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, Ottawa, Canada, June 2006.

[PCS05] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 218–232, Barcelona, Spain, November 2005. Springer-Verlag.

[PD96] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996.

[Pie02] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PJEJ04] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Four generations of SPEC CPU benchmarks: what has changed and what has not. Technical Report TR-041026-01-1, University of Texas Austin, 2004.

[Pop06] S. Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. PhD thesis, École Nationale Supérieure des Mines de Paris, dec 2006.

[Pot96] F. Pottier. Simplifying subtyping constraints. In *ACM Intl. Conf. on Functional Programming (ICFP'96)*, volume 31(6), pages 122–133, 1996.

[PS99] M. Pelletier and J. Sakarovitch. On the representation of finite deterministic 2-tape automata. *Theoretical Computer Science*, 225(1-2):1–63, 1999.

[PSX+04] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[PTCV04] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *ACM Supercomputing'04*, Pittsburgh, Pennsylvania, November 2004. 15 pages.

[PTV02] D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance? matrix-multiply revisited. In *SuperComputing'02*, Baltimore, Maryland, November 2002.

[Pug91a] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE Conf. on Supercomputing*, pages 4–13, Albuquerque, August 1991.

[Pug91b] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, August 1991.

[Pug91c] W. Pugh. Uniform techniques for loop optimization. In *ACM Intl. Conf. on Supercomputing (ICS'91)*, pages 341–352, Cologne, Germany, June 1991.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.

[QR99] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. Technical Report 1228, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, France, January 1999.

[QRW00] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, October 2000.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symp. on Principles of Programming Languages (PoPL'95)*, San Francisco, CA, January 1995.

[RP99] L. Rauchwerger and D. Padua. The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, 10(2):160–180, 1999.

[RRH03] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Intl. J. of Parallel Programming*, 31(4):251–283, 2003.

[RS97a] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1: Word Language Grammar. Springer-Verlag, 1997.

[RS97b] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 3: Beyond Words. Springer-Verlag, 1997.

[RZR04] S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France, 2004. IEEE Computer Society.

[SA05] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005)*. IEEE Computer Society, 2005.

[SAMO03] M. Stephenson, S. P. Amarasinghe, M. C. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, pages 77–90, San Diego, California, 2003.

[SAR$^+$00] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. Technical Report HPL-2000-31, Hewlett-Packard, May 2000.

[SCFS98] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independant storage mapping for loops. In *ACM Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, 8, 1998.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, UK, 1986.

[Smi00] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000. (Keynote Talk).

[SP81] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two Approaches to Interprocedural Data Flow Analysis. Prenticce Hall, 1981.

[Spe] Standard performance evaluation corp. `http://www.spec.org`.

[SPHC02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[SRW99] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symp. on Principles of Programming Languages (PoPL'99)*, pages 105–118, San Antonio, Texas, January 1999.

[TFJ86] R. Triolet, P. Feautrier, and P. Jouvelot. Automatic parallelization of fortran programs in the presence of procedure calls. In *Proc. of the 1stEuropean Symp. on Programming (ESOP'86)*, number 213 in LNCS, pages 210–222. Springer-Verlag, March 1986.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Intl. Conf. on Compiler Construction*, Grenoble, France, April 2002.

[TP93] P. Tu and D. Padua. Automatic array privatization. In *6thLanguages and Compilers for Parallel Computing*, number 768 in LNCS, pages 500–521, Portland, Oregon, August 1993.

[TS85] J.-P. Tremblay and P.-G. Sorenson. *The theory and practice of compiler writing*. McGraw-Hill, 1985.

[TVA05] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.

[TVSA01] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'01)*, pages 232–242, 2001.

[VAGL03] X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. PACT*, pages 68–78, 2003.

[VBC06] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.

[VBJC03] Sven Verdoolaege, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Multi-dimentsional incremetal loops fusion for data locality. In *ASAP*, pages 17–27, 2003.

[VCBG06] N. Vasilache, A. Cohen, C. Bastoul, and S. Girbal. Violated dependence analysis. In *ACM Intl. Conf. on Supercomputing (ICS'06)*, Cairns, Australia, June 2006.

[vE01] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'01)*, pages 118–132, 2001.

[VG98] T. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, October 1998.

[Vis01] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[Vui94] J. E. Vuillemin. On circuits and numbers. *IEEE Trans. on Computers*, 43(8):868–879, 1994.

[WCPH01] P. Wu, A. Cohen, D. Padua, and J. Hoeflinger. Monotonic evolution: an alternative to induction variable substitution for dependence analysis. In *ACM Intl. Conf. on Supercomputing (ICS'01)*, Sorrento, Italy, June 2001.

[Wol92] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992. Published as CSL-TR-92-538.

[Wol96] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[Won95] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.

[WPD00] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 2000.

[Xue94] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[YLR$^+$03] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, San Diego, CA, June 2003.

Résumé

La loi de Moore sur semi-conducteurs approche de sa fin. L'evolution de l'architecture de von Neumann à travers les 40 ans d'histoire du microprocesseur a conduit à des circuits d'une insoutenable complexité, à un très faible rendement de calcul par transistor, et une forte consommation énergetique. D'autre-part, le monde du calcul parallèle ne supporte pas la comparaison avec les niveaux de portabilité, d'accessibilité, de productivité et de fiabilité de l'ingénérie du logiciel séquentiel. Ce dangereux fossé se traduit par des défis passionnants pour la recherche en compilation et en langages de programmation pour le calcul à hautes performances, généraliste ou embarqué. Cette thèse motive notre piste pour relever ces défis, introduit nos principales directions de travail, et établit des perspectives de recherche.

Abstract

Moore's law on semiconductors is coming to an end. Scaling the von Neumann architecture over the 40 years of the microprocessor has led to unsustainable circuit complexity, very low compute-density, and high power consumption. On the other hand, parallel computing practices are nowhere close to the portability, accessibility, productivity and reliability levels of single-threaded software engineering. This dangerous gap translates into exciting challenges for compilation and programming language research in high-performance, general purpose and embedded computing. This thesis motivates our approach to these challenges, introduces our main directions and results, and draws research perspectives.