On the Decidability of Phase Ordering Problem in Optimizing Compilation

Iterative Compilation, Library Generation, Compiler Optimization Parameters Space Exploration and Beyond

Sid-Ahmed-Ali TOUATI
University of Versailles.
PRiSM Laboratory.
45 avenue des Etats Unis.
78035 Versailles cedex. France.
touati@prism.uvsq.fr

Denis BARTHOU
University of Versailles.
PRiSM Laboratory.
45 avenue des Etats Unis.
78035 Versailles cedex. France.
Denis.Barthou@prism.uvsq.fr

Categories and Subject Descriptors

D [**3**]: 4

General Terms

Algorithms, Performance, Theory.

Keywords

Phase Ordering, Parameters Space Exploration, Iterative Compilation, Library Generation, Optimizing Compilation.

ABSTRACT

We are interested in the computing frontier around an essential question about compiler construction: having a program $\mathcal{P}$ and a set $\mathcal{M}$ of non parametric compiler optimization modules (called also phases), is it possible to find a sequence s of these phases such that the performance (execution time for instance) of the final generated program $\mathcal{P}'$ is "optimal" ? We prove in this article that this problem is undecidable in two general schemes of optimizing compilation: iterative compilation and library optimization/generation. Fortunately, we give some simplified cases when this problem becomes decidable, and we provide some algorithms (not necessary efficient) that can answer our main question.

Another essential question that we are interested in is parameters space exploration in optimizing compilation (tuning optimizing compilation parameters). In this case, we assume a fixed sequence of optimization, but each optimization phase is allowed to have a parameter. We try to figure out how to compute the best parameter values for all program transformations when the compilation sequence is given. We also prove that this general problem is undecidable and we provide some simplified decidable instances.

CF'06, May 3–5, 2006, Ischia, Italy.

1. INTRODUCTION

The notion of an "optimal" program is sometimes ambiguous in optimizing compilation. Using an absolute definition, an optimal program $\mathcal{P}^*$ means that there is no other equivalent program $\mathcal{P}$ faster than $\mathcal{P}^*$, whatever be the input data. This is equivalent to state that the optimal program should run as fast as the longest dependence chain in its trace. This notion of optimality cannot exist in practice: Schwiegelshohn *et al* showed in [20] that there are loops with conditional jumps for which no semantically equivalent time-optimal program exists on parallel machines, even with speculative execution[1]. More precisely, they showed why it is impossible to write a program that is the fastest for any input data. This is because the presence of conditional jumps makes the program execution paths dependent on the input data, so it is not guaranteed that a program shown faster for a considered input data set (*i.e.*, for a given execution path) remains the fastest for all possible input data. Furthermore, Schwiegelshohn *et al* convinced us that "optimal" codes for loops with branches (with arbitrary input data) requires the ability to express and execute a program with unbounded speculative window. Since any real speculative feature is limited in practice[2], it is "impossible" to write an optimal code for some loops with branches on real machines.

In our work, we define the program optimality according to the input data. So, we say that a program $\mathcal{P}^*$ is optimal if there is not another equivalent program $\mathcal{P}$ faster than $\mathcal{P}^*$ considering the same input data. Of course, the optimal program $\mathcal{P}^*$ related to the considered input data I^* must still execute correctly for any other input data, but not necessarily in the fastest speed of execution. In other term, we do not try to build efficient specialized programs, i.e., we should not generate programs that execute only for a certain input data set. Otherwise a simple program that only prints the results would be sufficient for fixed input data.

With this notion of optimality, we can ask the general question: how to build a compiler that generates an optimal program given an input data set ? Such question is very difficult to answer, since we are not able till now to enumerate all the possible automatic program rewriting methods in compilation (some are present in the

[1]Indeed, the cited paper does not contain a formal full proof as claimed by the authors, but a persuasive reasoning.
[2]If the speculation is static, the code size is finite. If speculation is made dynamic, the hardware speculative window is bounded.

literature, others have to be set up in the future). So, we first address in this work another similar question: given a finite set $\mathcal{M}$ of compiler optimization modules, how to build an automatic method to combine them in a finite sequence that produces an optimal program? We mean by compiler optimization module a program transformation that rewrites the original code. Unless they are encapsulated inside code optimization modules, we exclude program analysis passes since they do not modify the code.

This paper provides a formalism for some general questions about phase ordering. Our formal writing allows us to give preliminary answers from the computer science perspective about decidability (what we can really do by automatic computation) and undecidability (what we can never do by automatic computation). We will show that our answers are tightly correlated to the nature of the models (functions) used to predict or evaluate the programs performances. Note that we are not interested in the efficiency aspects of compilation and code optimization: we know that most of the code optimization problems are inherently NP-complete. Consequently, the proposed algorithms in this paper are not necessarily efficient, and are written for the purpose of demonstrating the decidability of some problems. Proposing efficient algorithms for decidable problems is another research aspect outside the current scope.

This paper is organized as follows. Section 2 gives a short overview about some phase ordering studies in the literature, as well as some performance prediction modeling. Section 3 defines a formal model for the phase ordering problem that allows us to prove some negative decidability results. Next, in Section 4, we show some general optimizing compilation scheme in which the phase ordering problem becomes decidable. Section 5 explores the problem of tuning optimizing compilation parameters with a compilation sequence. Finally, we present our future work before concluding.

2. BACKGROUND

The problem of phase ordering in optimizing compilation is coupled to the problem of performance modeling, since the performance prediction/estimation may guide the search process. The two following subsections present a quick overview of related work.

2.1 Performance Modeling and Prediction

Program performance modeling and estimation on a certain machine is an old (and is still) an important research topic aiming to guide code optimization. The simplest performance prediction formula is the linear function that computes the execution time of a sequential program on a simple von-Neumann machine: it is simply a linear function of the number of executed instructions. With the introduction of memory hierarchy, parallelism at many level (instructions, threads, process), branch prediction and speculation, performance prediction becomes more complex than a simple linear formula. The exact *shape* or the nature of such function and the parameters that it involves are two unknown problems until now. However, there exist many articles that try to define approximated performance prediction functions:

- *Statistical Linear Regression Models:* the parameters involved in the linear regression are usually chosen by the authors. Many program executions or simulation through multiple data sets allow to build statistics that compute the coefficients of the model [21, 6].

- *Static Algorithmic Models:* usually, such models are algorithmic analysis methods that try to predict a program performance [4, 16, 25, 23]. For instance, the algorithm counts the instructions of a certain type, or makes a guess of the local instruction schedule, or analyzes data dependencies to predict the longest execution path, etc.

- *Comparison Models:* instead of predicting a precise performance metric, some studies provide models that compare two code versions and try to predict the fastest one [11, 24].

Of course, the best and the most accurate performance prediction is the Turing machine itself, since it executes the program and hence we can directly measure the performance. This is what is usually used in iterative compilation and library generation for instance.

The main problem with performance prediction models is their aptitude to reflect the real performance on the real machine. As well explained by Rai Jain [18], the common mistake in statistical modeling is to trust a model simply because it plots a *similar* curve compared to the real plot (a proof by eyes !). Indeed, this sort of experimental validation is not correct from the statistical science theory, and there exist formal statistical methods [18] that check if a model fits the reality. Until now, we have not found any study that validates a program performance prediction model using such formal statistical methods.

2.2 Some Attempts in Phase Ordering

Finding the best order in optimizing compilation is an old problem. The most common case is the dependence between register allocation and instruction scheduling in instruction level parallelism processors as shown in [7]. Many other cases of inter-phase dependencies exist, but it is hard to analyze all the possible interactions [26].

Click and Cooper in [3] present a formal method that combines two compiler modules to build a *super*-module that produces better (faster) programs than if we apply each module separately. However, they do not succeed to generalize their framework of module combination, since they prove it for only two special cases, which are constant propagation and dead code elimination.

In [12], the authors use exhaustive enumeration of possible compilation sequences (restricted to a limited sequence size). They try to find if any "best" compilation sequence emerges. The experimental results show that, unfortunately, there is not a winning compilation sequence. We think that this is because such compilation sequence depends not only on the compiled program, but also on the input data and the underlying executing machine.

In [22], the authors target a similar objective as in [3]. They succeed to produce *super*-modules that guarantee performance optimization. However, they combine two analysis passes followed by a unique program rewriting phase. In our work, we try to find the best combination of code optimization modules, excluding program analysis passes (unless they belong to the code transformation modules).

In [15], the authors evaluate by using a performance model the different optimization sequences to apply to a given program. The model determines the profit of optimization sequences according to register resource and cache behavior. Optimizations consider only scalars and the same optimizations are applied whatever be the values of the inputs. In our article, we assume on the contrary that the optimization sequence should depend on the value of the input (in order to be able to speak about the optimality of a program).

Finally, there is the whole field of iterative compilation. In this research activity, looking for a good compilation sequence requires to compile the program multiple times iteratively, and at each it-

eration, a new code optimization sequence is used [5, 24] until a "good" solution is reached. In such frameworks, any kind of code optimization can be sequenced, the program performance may be predicted or accurately computed via execution or simulation. There exist other attempts that try to combine a sequence of high level loop transformations [1, 13]. As mentioned, such methods are devoted to regular high performance codes and only use loop transformation in the polyhedral model.

In this paper, we give a general formalism for the phase ordering problem and its multiple variants that incorporate the work presented in this section.

3. TOWARDS A THEORETICAL MODEL FOR PHASE ORDERING PROBLEM

In this section, we give our theoretical framework about the phase ordering problem. Let $\mathcal{M}$ be a finite set of program transformations. We would like to construct an algorithm $\mathcal{A}$ that has three inputs: a program $\mathcal{P}$, an input data I and a desired execution time T for the transformed program. For each input program and its input data set, the algorithm $\mathcal{A}$ must compute a finite sequence $s = m_n \circ m_{n-1} \circ \cdots \circ m_0$, $m_i \in \mathcal{M}^*$ of optimization modules[3]. The same transformation can appear multiple times in the sequence, as it occurs already in real compilers (for constant propagation/dead code elimination for instance). If s is applied to $\mathcal{P}$, it must generate an optimal transformed program $\mathcal{P}^*$ according to the input data I. Each optimization module $m_i \in \mathcal{M}$ has a unique input which is the program to be rewritten, and has an output $\mathcal{P}' = m_i(\mathcal{P})$. So, the final generated program $\mathcal{P}^*$ is $(m_n \circ m_{n-1} \circ \cdots \circ m_0)(\mathcal{P})$.

We must have a clear concept and definition of a program transformation module. Nowadays, many optimization techniques are complex toolboxes with many parameters. For instance, loop unrolling and loop blocking require a parameter which is the degree of unrolling or blocking. Until Section 5, we do not consider such parameters in our formal problem. We handle them by considering, for each program transformation, a finite set of parameter values, which is the case in practice. Therefore loop unrolling with an unrolling degree of 4 and loop unrolling with a degree of 8 are considered as two different optimizations. Given such finite set of parameter values per program transformation, we can define a new compilation module for each pair of program transformation and parameter value. So, for the remainder of the text (until Section 5), a program transformation can be considered as a module without any parameter except the program to be optimized.

In order to check that the execution time has reached some value T, we assume that there is a performance evaluation function t that allows to precisely evaluate or predict the execution time (or other performance metrics) of a program $\mathcal{P}$ according to the input data I. Let $t(\mathcal{P}, I)$ be the predicted execution time. Thus, t can predict the execution time of any transformed program $\mathcal{P}' = m(\mathcal{P})$ when applying a program transformation c. If we apply a sequence of program transformations, t is assumed to be able to predict the execution time of the final transformed program, *i.e.*, $t(\mathcal{P}', I) = t((m_n \circ m_{n-1} \circ \cdots \circ m_0)(\mathcal{P}), I)$. t can be either the measure of performance on the real machine, obtained through execution of the program with its inputs, a simulator or a performance model. In this article, we do not make the distinction between the three cases and assume that t is an arbitrary computable function. Next, we give a formal description of the phase ordering problem in optimizing compilation.

[3] $\circ$ denotes the symbol of function combination (concatenation).

PB. 1 (PHASE-ORDERING). *Let t be an arbitrary performance evaluation function. Let $\mathcal{M}$ be a finite set of program transformations. $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall \mathcal{P}$ a program, $\forall I$ input data, does there exist a sequence $s \in \mathcal{M}^*$ such that $t(s(\mathcal{P}), I) < T$? In other words, if we define the set:*

$$S_{t,\mathcal{M}}(\mathcal{P}, I, T) = \{s \in \mathcal{M}^* | t(s(\mathcal{P}), I) < T\}$$

is the set $S_{t,\mathcal{M}}(\mathcal{P}, I, T)$ empty?

Textually, the phase ordering problem tries to determine whether there exists or not a compilation sequence s which results in an execution time (for the transformed program with input data I) lower than a bound T.

If there is an algorithm that decides the phase ordering problem, then there is an algorithm that computes one sequence s such that $t(s(\mathcal{P}), I) < T$, provided that t always terminates. Indeed, enumerating the code optimization sequences in lexicographic order always finds an admissible solution to Problem 1. Deciding the phase ordering problem is therefore the key for finding the best optimization sequence.

3.1 Decidability Results

In our problem formulation, we assume the following characteristics:

1. t is a computable function. $t(\mathcal{P}, I)$ terminates when $\mathcal{P}$ terminates on the input I. This definition is compatible with the fact that t can be the measured execution time on a real machine;
2. each program transformation $m \in \mathcal{M}$ is computable, always terminates and preserves the program semantics;
3. program $\mathcal{P}$ always terminates;
4. the final transformed program $\mathcal{P}' = s(\mathcal{P})$ executes at least one instruction, *i.e.*, the final execution time is strictly positive.

The phase ordering problem corresponds to what occurs in a compiler: whatever the program and input be given by the user (if the compiler resorts to profiling), the compiler has to find a sequence of optimizations reaching some (not very well defined) performance threshold. Answering the question of the phase ordering problem as defined in Problem 1 depends on the performance prediction model t. Since the function (or its class) t is not defined, Problem 1 cannot be answered as it is, and requires to have another formulation that slightly changes its nature. We consider in this paper a modified version, where the function t is not known by the optimizer.

PB. 2 (MODIFIED PHASE-ORDERING). *Let $\mathcal{M}$ be a finite set of program transformations. For any performance evaluation function t, $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall \mathcal{P}$ a program, $\forall I$ input data, does there exist a sequence $s \in \mathcal{M}^*$ such that $t(s(\mathcal{P}), I) < T$? In other words, if we define the set:*

$$S_{\mathcal{M}}(t, \mathcal{P}, I, T) = \{s \in \mathcal{M}^* | t(s(\mathcal{P}), I) < T\},$$

is the set $S_{\mathcal{M}}(t, \mathcal{P}, I, T)$ empty?

This problem corresponds to the case where t is not an *approximates* model but is the real executing machine (the most precise model). Let us present the intuition behind this statement: a compiler always has an architecture model of the target machine (resource constraints, instruction set, general architecture, latencies of

caches,...). This model is assumed to be correct (meaning that the real machine conforms according to the model) but does not take into account all mechanisms of the hardware. Thus in theory, an infinite number of different machines fit into the model, and we must assume the real machine is any of them. As the architecture model is incomplete and performance also depends usually on non-modeled features (conflict misses, data alignment, operation bypasses,...), the performance evaluation model of the compiler is incorrect. This suggests that the performance evaluation function of the real machine can be any performance evaluation function, even if there is a partial architectural description of this machine. Consequently, Problem 2 corresponds to the case of the phase ordering problem when t is the most precise performance model which is the real executing machine (or simulator): the real machine measures the performance of its own executing program (for instance, by using its internal clock or its hardware performance counters).

In the following lemma, we assume an additional hypothesis: there exists a program that can be optimized into an infinite number of different programs. This necessarily requires that there is an infinite number of different optimization sequences. But this is not sufficient. As sequences of optimizations in $\mathcal{M}$ are considered as words made of letters from the alphabet $\mathcal{M}$, the set of sequences is always infinite, even with only one optimization in $\mathcal{M}$. For instance, fusion and loop distribution can be used repetitively to build sequences as long as desired. However, this infinite set of sequences will only generate a finite number of different optimized codes (ranging from all fusioned loops, to all distributed loops). If the total number of possible generated programs is bounded, then it may be possible to fully generate them in a bounded compilation time: it is therefore easy to check the performance of every generated program and to keep the best one. In our hypothesis, we assume that the set of all possible generated programs (generated using the distinct compilation sequences belonging to $\mathcal{M}^*$) is infinite. One simple optimization such as strip-mine, applied many times to a loop with parametric bounds, generates as many different programs. Likewise, unrolling a loop with parametric bounds can be performed an infinite number of times. Note that the decidability of Problem 2 when the cardinality of $\mathcal{M}^*$ is infinite while the set of distinct generated programs is finite remains an open problem.

LEMMA 1. *Modified Phase-Ordering is an undecidable problem if there exists a program that can be optimized into an infinite number of different programs.*

PROOF. The proof works by reduction of the following problem:

Problem (Empty Set) *Given L a recursive enumerable language, is L empty ?*

This problem is known undecidable by application of Rice theorem [9].

We first reformulate the phase-ordering problem as a problem on computable functions instead of a problem on optimization sequences and evaluation functions. Sequences s of optimizations in $\mathcal{M}$ are considered as words made of letters from the alphabet $\mathcal{M}$. There exists a program $\mathcal{P}_0$ such that optimization sequences applied to $\mathcal{P}_0$ generate an infinite number of different programs. Thus, there exists an algorithm $a_{\mathcal{M},\mathcal{P}_0}$ that, given an integer i, enumerates optimization sequences in lexicographical order, finds the first i sequences s_i that generate different optimized programs and outputs $s_i(\mathcal{P}_0)$. $a_{\mathcal{M},\mathcal{P}_0}(i) = s_i(\mathcal{P}_0)$. This function is a bijective mapping between integers and optimized versions of $\mathcal{P}_0$. Now, we can define, for any evaluation function t, a function t': $t'(m, I) = t(a_{\mathcal{M},\mathcal{P}_0}(m), I)$ for all integers m, I. t' is a computable function that always terminate. This defines a bijective mapping between evaluation functions and terminating computable functions with two parameters. Only one of these parameters (m) is not an input of the problem. We introduce one more in order to be able to perform the reduction: as there exists a bijective function h mapping any pair of integers to an integer, we define a mapping between evaluation functions and terminating computable functions with three parameters: $t''(m, n, I) = t(a_{\mathcal{M},\mathcal{P}_0}(h(m, n)), I)$. The Modified Phase-ordering problem is equivalent to deciding whether the set:

$$S'(t'', I, T) = \{(m, n) | t''(m, n, I) < T\}$$

is empty or not, where t'' is any computable function that always terminate.

Now, we build the reduction. Consider a recursive enumerable language L. There exists a computable function g in $\{0, 1\}$ such that $m \in L \Leftrightarrow g(m) = 1$. We build the following function $t_{g,T}$, based on g:

- Input: m, n and I
- Perform at most n steps of the computation of $g(m)$.
- If the computation has finished and $g(m) = 1$ then return $T - 1$
- Else return $T + 1$.

This function $t_{g,T}$ always terminates and is computable. Moreover, there exists n such that $t_{g,T}(m, n, I) < T$ iff $g(m) = 1$.

Now, given some integer T and an input I:

- If $S'(t_{g,T}, I, T)$ is not empty, then there exist m and n such that $g(m)$ stops before n computation steps and $g(m) = 1$. Therefore L is not empty.
- If $S'(t_{g,T}, I, T)$ is empty, then for all integer values m and n, $t_{g,T}(m, n, I) \geq T$. According to the definition of $t_{g,T}$, it means that for all m and n, either g does not stop before n steps of computation, or g stops and $g(m) = 0$. That implies that for all m integer, $m \notin L$: $L = \emptyset$.

This shows that there is a reduction from the problem Empty Set to the Modified Phase Ordering problem. As the problem Empty Set is undecidable, the modified phase ordering problem is also undecidable. □

We provide here a variation on the modified phase ordering problem that corresponds to the library optimization issue: program and (possibly) inputs are known at compile-time, but the optimizer has to adapt its sequence of optimization to the underlying architecture/compiler. This is what happens in Spiral [14] and FFTW [8]. If the input is also part of the unknowns, the problem has the same difficulty.

PB. 3 (PHASE ORDERING FOR LIBRARY OPTIMIZATION). *Let $\mathcal{M}$ be a finite set of program transformations, $\mathcal{P}$ the program of a library function, I some input and T an execution time. For any performance evaluation function t, does there exist a sequence $s \in \mathcal{M}^*$ such that $t(s(\mathcal{P}), I) < T$? In other words, if we define the set:*

$$S_{\mathcal{P},I,\mathcal{M},T}(t) = \{s \in \mathcal{M}^* | t(s(\mathcal{P}), I) < T\}$$

is the set $S_{\mathcal{P},I,\mathcal{M},T}(t)$ empty?

The decidability results of Problem 3 are stronger than those of Problem 2: here the compiler knows the program, its inputs, the

optimizations to play with and the performance bound to reach. However, there is still no algorithm to find out the best optimization sequence, if the optimizations may generate a infinite number of different program versions.

LEMMA 2. *Phase Ordering for library optimization is undecidable if optimizations can generate an infinite number of different programs for the library functions.*

PROOF. The proof is the same as the previous one, as the proof does not depend neither on the input I nor on the bound T. □

The next section gives other formulations of the Phase-Ordering problem that do not alter the decidability results proved in this section.

3.2 Another formulation of Phase Ordering Problem

Instead of having a function that predicts the execution time, we can consider a function g that predicts the performance gain or speedup. g would be a function with three inputs: the input program $\mathcal{P}$, the input data I and a transformation module $m \in \mathcal{M}$. The performance prediction function $g(\mathcal{P}, I, m)$ computes the performance gain if we transform the program $\mathcal{P}$ to $m(\mathcal{P})$ and by considering the same input data I. For a sequence $s = (m_n \circ m_{n-1} \cdots \circ m_0) \in \mathcal{M}^*$ we define the gain $g(\mathcal{P}, I, s) = g(\mathcal{P}, I, m_0) \times g(m_0(\mathcal{P}), I, m_1) \times \cdots \times g((m_{n-1} \circ \cdots \circ m_0)(\mathcal{P}), I, m_n)$. Note that, since the gains (and speedups) are fractions, the whole gain of the final generated program is the product of the partial intermediate gains. The ordering problem in this case becomes the problem of computing a compilation sequence that results in a maximal speedup, formally written as follows. This problem formulation is equivalent to the initial one that tries to optimize the execution time instead of speedup.

PB. 4 (MODIFIED PHASE-ORDERING WITH PERF. GAIN). *Let $\mathcal{M}$ be a finite set of program transformations. For any performance gain function g, $\forall k \in \mathbb{N}$ a performance gain, $\forall \mathcal{P}$ a program, $\forall I$ input data, does there exist a sequence $s \in \mathcal{M}^*$ such that $g(\mathcal{P}, I, s) \geq k$? In other words, if we define the set:*

$$S_{\mathcal{M}}(g, \mathcal{P}, I, k) = \{s \in \mathcal{M}^* | g(\mathcal{P}, I, s) \geq k\},$$

is the set $S_{\mathcal{M}}(g, \mathcal{P}, I, k)$ empty?

We can easily see that Problem 2 is equivalent to Problem 4. This is because g and t are dependent each other by the following usual equation of performance gain:

$$g(\mathcal{P}, I, m) = \frac{t(\mathcal{P}, I) - t(m(\mathcal{P}), I)}{t(\mathcal{P}, I)}$$

4. EXAMPLES OF DECIDABLE SIMPLIFIED CASES

In this section we give some decidable instances of the phase ordering problem. As a first case, we define another formulation of the problem that introduces a monotonic cost function. This formulation models the real existing compilation approaches. As a second case, we model generative compilation and show that phase ordering is decidable in this case.

4.1 Models with Compilation Costs

In Section 3, the phase ordering problem is defined using a performance evaluation function. In this section, we add another function c that models a cost. Such cost may be the compilation time, the number of distinct compilation passes inside a compilation sequence, the length of a compilation sequence, distinct explored compilation sequences, etc. The cost function has two inputs: the program $\mathcal{P}$ and a transformation pass m. Thus, $c(\mathcal{P}, m)$ gives the cost of transforming the program $\mathcal{P}$ to $\mathcal{P}' = m(P)$. Such cost does not depend on input data I. The phase ordering problem including the cost function becomes the problem of computing the best compilation sequence with a bounded cost.

PB. 5 (PHASE-ORDERING WITH DISCRETE COST FUNCTION). *Let t be performance evaluation function that predicts the execution time of any program $\mathcal{P}$ given input data I. Let $\mathcal{M}$ be a finite set of optimization modules. Let $c(\mathcal{P}, m)$ be a function that computes the cost of transforming the program $\mathcal{P}$ to $\mathcal{P}' = m(P)$, $m \in \mathcal{M}$. Does there exist an algorithm $\mathcal{A}$ that solves the following problem ? $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall K \in \mathbb{N}$ a compilation cost, $\forall \mathcal{P}$ a program, $\forall I$ input data, compute $\mathcal{A}(\mathcal{P}, I, T) = s$ such that $s = (m_n \circ m_{n-1} \cdots \circ m_0) \in \mathcal{M}^*$ and $t(s(\mathcal{P}), I) < T$ with $c(\mathcal{P}, m_0) + c(m_0(\mathcal{P}), m_1) + \cdots + c((m_{n-1} \circ \cdots \circ m_0)(\mathcal{P}), m_n) \leq K$.*

We see in this section that if the cost function c is a strictly increasing function, then we can provide a recursive algorithm that solves Problem 5. First, we define the monotonic characteristics of the function c. We say that c is strictly increasing iff

$$\forall m, m' \in \mathcal{M}, \quad c(\mathcal{P}, m) < c(s(\mathcal{P}), m')$$

That is, applying a program transformation sequence $m_n \circ m_{n-1} \cdots \circ m_0 \in \mathcal{M}^*$ to a program $\mathcal{P}$ has always a higher integer cost than applying $m_{n-1} \cdots \circ m_0 \in \mathcal{M}^*$. Such assumption is true for the case of function costs such as compilation time[4], number of compilation passes, etc. Each practical compiler uses an implicit cost function.

Building an algorithm that computes the best compiler optimization sequence given a strictly increasing cost function is an easy problem because we can use an exhaustive search of all possible compilation sequences with bounded cost. Algorithm 1 provides a trivial recursive method: it first looks for all possible compilation sequences under the considered cost, then it iterates over all these compilation sequences to check whether we could generate a program with the bounded execution time. Such process terminates because we are sure that the cumulative integer costs of the intermediate program transformations will certainly reach the limit K.

As illustration, the work presented in [12] belongs to this family of decidable problems. Indeed, the authors compute all possible compilation phase sequences, but by restricting themselves to a given number of phases in each sequence. Such number is modeled in our framework as a cost function defined as follows: $\forall \mathcal{P}$ a program ,

$$c(\mathcal{P}, s) = \begin{cases} 1 + c(\mathcal{P}, (m_{n-1} \circ \cdots \circ m_0)) & \forall (m_n \circ \cdots \circ m_0) \in \mathcal{M}^* \\ 1 & \forall m \in \mathcal{M} \end{cases}$$

Textually it means that we associate to each compilation sequence the cost which is simply equal to the number of phases inside the compilation sequence. The authors in [12] limit the number of phases (to 10 or 15 as example). Consequently, the number of possible combinations becomes bounded which makes the problem of phase ordering decidable. Algorithm 1 can be used to generate the best compilation sequence if we consider a cost function as a fixed

[4]The time on an executing machine is discrete since we have clock cycles.

Algorithm 1 Computing a Good Compilation Sequence in the Compilation Cost Model

Require: a program P
Require: a cost $K \in \mathbb{N}$
Require: an execution time $T \in \mathbb{N}$
Require: $\hat{1}$ a neutral optimization: $\hat{1}(P) = P \wedge c(P, \hat{1}) = 0$
  /* we first compute the SET of all possible compilation sequences under the cost limit K */
  $SET \leftarrow \{\hat{1}\}$
  $stop \leftarrow false$
  while $\neg stop$ **do**
    $stop \leftarrow true$
    for all $s \in SET$ **do**
      $visited[s] \leftarrow false$
    end for
    for all $s \in SET$ **do**
      if $\neg visited[s]$ **then**
        for all $m_i \in \mathcal{M}$ **do** {for each compilation phase}
          if $c(P, s \circ m_i) \leq K$ **then** {save a new compilation sequence with a bounded cost if the cost is bounded by K}
            $SET \leftarrow SET \cup \{s \circ m_i\}$
            $stop \leftarrow false$
          end if
        end for
      end if
      $visited[s] \leftarrow true$
    end for
  end while
  /* now, we look for a compilation sequence that produces a program with the bounded execution time */
  $exists\_solution \leftarrow false$
  for all $s \in SET$ **do**
    if $t(P, s) \leq T$ **then**
      $exists\_solution \leftarrow true$
      $return\ s$
    end if
  end for
  if $\neg exists\_solution$ **then**
    print "No solution exists to Problem 5"
  end if

number of phases.

The next section presents another simplified case in phase ordering, which is generative compilation.

4.2 One-Pass Generative Compilers

Generative compilation is a subclass of iterative compilation. In such simplified classes of compilers, the code of an intermediate program is optimized and generated in a one pass traversal of the abstract syntax tree. Each program part is treated and translated to a final code without any possible backtracking in the code optimization process. For instance, we can take the case of a program given as an abstract syntax tree. A set of compilation phases treats each program part, *i.e.* each sub-tree, and generates a native code for such part. Another code optimization module can no longer re-optimize the already generated program part, since any optimization module in generative compilation takes as input only program parts in intermediate form. When a native code generation for a program part is carried out, there is no way to re-optimize such program portion, and the process continues for other sub-trees until

Algorithm 2 Optimize_Node(n)

Require: an abstract syntax tree with root n
Require: a finite set of program transformations $\mathcal{M}$
  if n is not leaf **then**
    for all u child of n **do**
      $Optimize\_Node(u)$
    end for
    /*Generate all possible codes and choose the best one*/
    $best \leftarrow \phi$ {best code optimization}
    $time \leftarrow \infty$ {best performance}
    for all $s \in \mathcal{M}$ **do**
      if $t(n, s) \leq time$ **then**
        $best \leftarrow s$
        $time \leftarrow t(n, s)$
      end if
    end for
    apply the $best$ transformation to the node n without changing any child
  else {Generate all possible codes and choose the best one}
    $best \leftarrow \phi$ {best code optimization}
    $time \leftarrow \infty$ {best performance}
    for all $s \in \mathcal{M}$ **do**
      if $t(n, s) \leq time$ **then**
        $best \leftarrow s$
        $time \leftarrow t(n, s)$
      end if
    end for
    apply the $best$ transformation to the node n
  end if

finishing the whole tree. Note that the optimization process for each sub-tree is applied by a finite set of program transformations. In other words, generative compilers look for local "optimized" code instead of a global optimized program.

Given a program represented as an abstract syntax tree, the program optimization process computes the best compilation phase locally to each node, starting from leaves. For example, the SPIRAL project in [14] is a generative compiler. It performs a local optimization to each node as shown by Algorithm 2. It recursively applies a set of program transformations at each node, starting from the leaves. Each node is transformed locally without any consideration of its children.

As can be seen, finding a compilation sequence in generative compilation that produces the fastest program is a decidable problem (Algorithm 2). Since the size of intermediate representation forms decreases at each local application of program transformation, we are sure that the process of program optimization terminates when all intermediate forms have been transformed to native codes. In other terms, the number of possible distinct passes on a program becomes finite and bounded as shown in Algorithm 2: for each node of the abstract syntax tree, we apply locally a single code optimization (we iterate over all possible code optimization modules and we pick up the one that produces the best performance according to the chosen performance model). Furthermore, no code optimization sequence is searched locally (only a single pass is applied). Thus, if the total number of nodes in the abstract syntax tree is equal to $\tilde{n}$, then the total number of applied compilation sequences does not exceed $|\mathcal{M}| \times \tilde{n}$.

Of course, the decidability of one-pass generative compilers does not prevent them from having potentially high complexity: each local code optimization may be exponential (if it tackles NP-complete

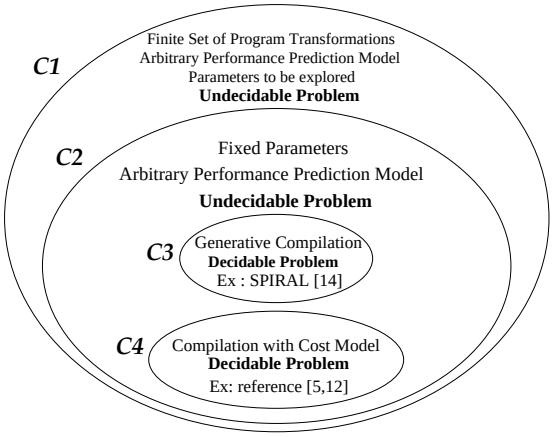

Figure 1: Classes of Phase-Ordering Problems

problem for instance). The decidability result only proves that, if we have a high computation power, we know that we can compute the optimal code after a bounded compilation time (possibly high).

This first part of the article investigates the decidability problem of phase ordering in optimizing compilation. Figure 1 synthesizes a whole view of the different classes of the investigated problems with their decidability results. The largest class of the phase ordering problem that we consider, denoted by C_1, assumes a finite set of program transformations with possible optimization parameters (to explore). If the performance prediction function is arbitrary, typically if it requires program execution or simulation, then this problem is undecidable. The second class of the phase ordering problem, denoted by $C_2 \subset C_1$, has the same hypothesis as C_1 except that the optimization parameters are fixed. The problem is undecidable too. However, we have identified two decidable classes of phase ordering problem which are C_3 and C_4 explained as follows. The class $C_3 \subset C_2$ considers one-pass generative compilation ; the program is taken as an abstract syntax tree (AST), and code optimization applies a unique local code optimization module on each node of the AST. The class $C_4 \subset C_2$ takes the same assumption as C_2 plus an additional constraint which is the presence of a cost model: if the cost model is a discrete increasing function, and if the cost of the code optimization is bounded, then C_4 is a class of decidable phase ordering problem.

The next section investigates another essential question in optimizing compilation, which is parameters space exploration.

5. COMPILER OPTIMIZATION PARAMETERS SPACE EXPLORATION

Nowadays, many compiler optimization methods are parametrized. For instance, loop unrolling requires an unrolling degree; loop blocking requires a blocking degree as well, etc. Actually, the complexity of phase ordering problem does not allow to explore jointly the the best sequence of the compilation steps and the best combinations of modules parameters. Usually, the community tries to find the "best" parameter combination when the compilation sequence is fixed. This section is devoted to study the decidability of such problem.

5.1 Towards a Theoretical Model

First, we suppose that we have $s \in \mathcal{M}^*$ a given sequence of optimizing modules belonging to a finite set $\mathcal{M}$. We assume that s is composed of n compilation sequences.

We associate for each optimization module $m_i \in \mathcal{M}$ a unique integer parameter $k_i \in \mathbb{N}$. The set of all parameters is grouped inside a vector $\overrightarrow{k} \in \mathbb{N}^n$, such that the i^{th} component of $\overrightarrow{k}$ is the parameter k_i of the m_i, the i^{th} module inside the considered sequence s. If the sequence s contains multiple instances of the same optimization module m, the parameter of each instance may have a distinct value from those of the other instances.

For a given program $\mathcal{P}$, applying a program transformation module $m \in \mathcal{M}$ requires a parameter value. Then, we write the transformed program as $\mathcal{P}' = m(\mathcal{P}, \overrightarrow{k})$.

As in the previous sections devoted to the phase ordering problem, we assume here the existence of a performance evaluation function t that predicts (or evaluates) the execution time of a program $\mathcal{P}$ having I as input data. We denote $t(\mathcal{P}, I)$ the predicted execution time. The formal problem of computing the best parameter values of a given set of program transformations in order to achieve the best performance can be written as follows.

PB. 6 (BEST-PARAMETERS). *Let t be a function that predicts the execution time of any program $\mathcal{P}$ given input data I. Let $\mathcal{M}$ be a finite set of program transformations and s a particular optimization sequence. Does there exist an algorithm $\mathcal{A}_{t,s}$ that solves the following problem ? $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall \mathcal{P}$ a program, $\forall I$ input data, $\mathcal{A}_{t,s}(\mathcal{P}, I, T) = \overrightarrow{k}$ such that $t(s(\mathcal{P}, \overrightarrow{k}), I) < T$.*

This general problem cannot be addressed as it is, since the answer depends on the shape of the function t. In this paper, we assume that the performance prediction function is built by an algorithm $\mathtt{a}$, taking s and $\mathcal{P}$ as parameters. Moreover, we assume the performance function $t = \mathtt{a}(\mathcal{P}, s)$ built by $\mathtt{a}$ takes $\overrightarrow{k}$ and I as parameters and is a polynomial function. Therefore, the performance of a program $\mathcal{P}$ with input I and optimization parameters $\overrightarrow{k}$ is $\mathtt{a}(\mathcal{P}, s)(I, \overrightarrow{k})$. We want to decide whether there are some parameters for the optimization modules that make the desired performance bound reachable:

PB. 7 (MODIFIED BEST-PARAMETERS). *Let $\mathcal{M}$ be a finite set of program transformations and s a particular optimization sequence of $\mathcal{M}^*$. Let $\mathtt{a}$ be an algorithm that builds a polynomial performance prediction function, according to a program and an optimization sequence. For all programs $\mathcal{P}$, for all inputs I and performance bound T, we define the set of parameters as:*

$$P_{s,t}(\mathcal{P}, I, T) = \{\overrightarrow{k} | \mathtt{a}(\mathcal{P}, s)(\overrightarrow{k}, I) < T\}.$$

Is $P_{s,t}(\mathcal{P}, I, T)$ empty ?

As noted earlier, choosing an appropriate performance model is a central decision to define whether Problem 6 is decidable or not. For instance, Problem 7 considers polynomial functions, which are a family of usual performance models (arbitrary linear regression models for instance). For instance, a three nested loop around a single operation has a complexity of $n_1 * n_2 * n_3$, if n_1, n_2 and n_3 are the respective loop trip counts of the three loops. With such a simple model, any polynomial can be generated. It is assumed that a realistic performance evaluation function would be as least as difficult as a polynomial function. Unfortunately, the following lemma

shows that if t is an arbitrary polynomial function, then Problem 7 is undecidable.

The following lemma states that Problem 7 is undecidable if there are at least 9 integer optimization parameters. In our context, this requires 9 optimizations in the optimizing sequence. Note that this number is constant when considering the best parameters, and is not a parameter itself (this is equal to the number of optimizations in the considered sequence). This number is fairly low compared to the number of optimizations found in state-of-the-art compilers (such as *gcc* or *icc* for instance). If there is only one parameter, then the problem is decidable. For a number of parameters between 2 and 8, the problem is still open [17] and Matiyasevich conjectured it as undecidable.

LEMMA 3. *The Modified Best-Parameters Problem is undecidable if the performance prediction function $t = \mathtt{a}(\mathcal{P}, s)$ is an arbitrary polynomial and if there are at least 9 integer optimization parameters.*

PROOF. The proof is based on a result published in 1982: given an arbitrary polynomial f with nine variables, Jones [10] proved that there is no recursive function which can determine whether f has a non-negative integer zero, in the sense that it finds an explicit zero or returns null otherwise.

Finding parameter values $\overrightarrow{k}$ such that, for an arbitrary polynomial p and for some given constant value I, $p(\overrightarrow{k}, I) < T$ is equivalent to finding the zeros of an arbitrary polynomial. Given a polynomial p, the polynomial $T * p(\overrightarrow{x}, I) * p(\overrightarrow{x}, I)$ reaches a value lower than T for some $\overrightarrow{x}$ only if $\overrightarrow{x}$ is a zero of p. This shows that to find the values for the bound is as difficult as finding the zeros of a polynomial. If $\mathtt{a}$ generates arbitrary polynomials, according to the value of $\mathcal{P}$, and there are at least 9 optimization parameters (I is not considered as a variable, as its value is constant), then Modified Best-Parameters Problem is undecidable. □

5.2 Examples of Simplified Decidable Cases

Our formal problem Best-Parameters is the formal writing of library optimizations. Indeed, in such area of program optimizations, the applications are given with a training data set. Then, people try to find the best parameter values of optimizing modules (inside a compiler usually with a given compilation sequence) that holds in the best performance. In this section, we show that some simplified instances of Best-Parameters problem becomes easily decidable. A first example is the OCEAN project [2], and a second one is the ATLAS framework [19].

The OCEAN project [2] optimizes a given program for a given data set by exploring all combinations of parameter values. Potentially, such value space is infinite. However, OCEAN restricts the exploration to finite set of parameter intervals. Consequently, the number of parameter combinations becomes finite, allowing a trivial exhaustive search of the best parameter values: each optimized program resulting from a particular value of the optimization parameters is generated and evaluated. The one performing best is chosen. Of course, if we use such exhaustive search, the optimizing compilation time become very high. So, one can provide efficient heuristics for exploring the bounded space of the parameters [24]. Currently, this is outside the scope of our article.

ATLAS [19] is another simplified case of the problem Best-Parameters. In the case of ATLAS, the optimization sequence is known, the programs to optimize are known (BLAS variants), and it is assumed that the performance does not depend on the value of the input (independence w.r.t. the matrix and vector values). Moreover, there is a performance model for the cache hierarchy (basically, the size of the cache) that, combined to the dynamic performance evaluation, limits the number of program executions (*i.e.*, performance evaluation) to do. For one level of cache and for matrix-matrix multiplication, there are three level of blocking controlled by three parameters, bounded by the cache size and a small number of loop interchange possibles. Exhaustive enumeration inside admissible values enable to find the best parameter value.

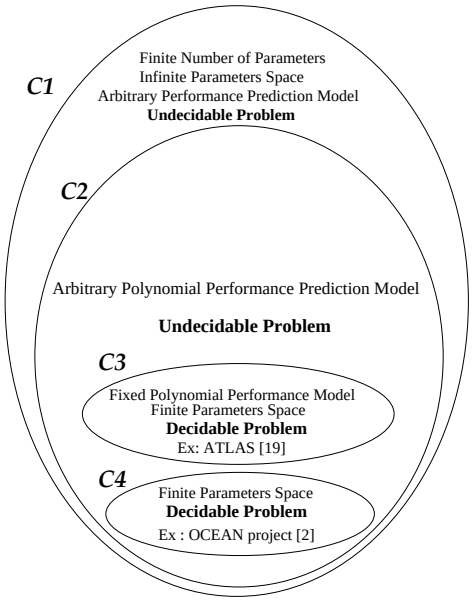

Figure 2: Classes of Best-Parameters Problems

Figure 2 synthesizes a whole view of the different classes of the investigated problems with their decidability results. The largest class of the best parameters exploration problem that we consider, denoted by C_1, assumes a finite set of optimization parameters with unbounded values (infinite space); The compiler optimization sequence is assumed fixed. If the performance prediction function is arbitrary, then this problem is undecidable. The second class of the best parameters exploration problem, denoted by $C_2 \subset C_1$, has the same hypothesis as C_1 except that the performance model is assumed as an arbitrary polynomial function. The problem is undecidable too. However, a trivial identified decidable class is the case of bounded (finite) parameters space. This is the case of the tools ATLAS (class C_3) and OCEAN (class C_4).

6. FUTURE WORK

The phase ordering problem studied in this article does not make any assumption about the kind or the family of the considered program transformations. Potentially, we can have an unbounded (but finite) number of optimizing modules inside a compiler, as long as they guarantee us the best performance. Consequently, the size of the compiler can be as large as we require. In a future work, we want to explore the phase ordering problem with an additional restriction which is the granted size to a compiler. For this purpose, and thanks to the results presented in [1], we will restrict the family of program transformations to the polyhedral ones. Indeed, the authors in [1] give a matrix coding of all polyhedral transformations and their possible combinations: the size of such matrix is finite and bounded, while its elements define all possible affine polyhedral program transformations. This matrix coding has the benefit for consuming a bounded space and allowing to ease the composi-

tion of program transformation.

Another future work to this article is to study the phase ordering problem with another kind of restriction. Instead of limiting the size of the optimizing compiler, we can put a limit on the size of the final transformed program. Does phase-ordering become decidable in this case ?

Finally, an important open problem remains the definition of a general family of performance prediction functions that makes the phase ordering problem decidable. In this paper, we proved that if such function requires the execution or the simulation of the considered program, then the phase ordering problem becomes undecidable. But what if the performance predictor does not require neither the execution nor the simulation of the program ? Of course, if the performance modeling is too trivial or too simple (see Section 4), it is highly probable that the phase ordering problem becomes decidable, but in this case the model would not fit the real program performance. So, we require to define a more general family of performance prediction functions, that are efficient enough to accurately model the real program performance while allowing to have a decidable phase ordering problem. As a first step, we will consider for instance linear regression models.

7. CONCLUSION

As far as we know, our article is the first formalisation of two known problems: the phase ordering in optimizing compilation and the compiler optimization parameters space exploration. Our article sets down the formal definition of the phase ordering problem in many compilation schemes such as static compilation, iterative compilation and library generation. Given an input data set for the considered program, the defined phase ordering problem is to find a sequence of code transformations (taken from a finite set of code optimizations) that increase the performance up to a fixed objective. Alternatively, we can consider too parametric code optimization modules, and then we can define the formal problem of best parameters space exploration. However in this case, the compilation sequence is fixed, and the searching process looks for the best code optimization parameters that increase the program performance up to a fixed objective.

We showed that the decidability of both these problems is tightly correlated to the function used to predict or to evaluate the program performance. If such function is an arbitrary polynomial function, or if it requires to execute a Turing machine (by simulation or by real execution on the considered underlying hardware), then both these problems are undecidable. This means that we can never have automatic solutions for them. We provided some simplified cases that make these problems decidable: for instance, we showed that if we include a compilation cost in the model (compilation time, number of generated programs, number of compilation sequences, etc.), then the phase ordering becomes obviously decidable. This is what all actual ad-hoc iterative compilation techniques really do. Also, we showed that if the parameters space is explicitly considered as bounded, then the best compiler parameter space exploration problem becomes trivially decidable too.

Our article proves then that the requirement to execute or to simulate a program is a major fundamental drawback for iterative compilation and for library generation in general. Indeed, they try to solve a problem that can never have an automatic solution. Consequently, it is impossible to bring a formal method that allows to accurately compare between the actual ad-hoc or practical methods of iterative compilation or for library generation[5, 12, 15, 24]. The experiments that can be made to highlight the efficiency of a method can never bring a guarantee that such iterative method would be efficient for other benchmarks. As a corollary, we can safely state that, since it is impossible to mathematically compare between iterative compilation methods (or between library generation tools) then we can consider that any proposed method is sufficiently “good” for only its set of experimented benchmarks and cannot be generalized as a concept or as a method.

Our article proves too that using iterative or dynamic methods for compilation is not fundamentally helpful for solving the general problem of code optimization. Such dynamic and iterative methods define distinct optimization problems that are unfortunately as undecidable as static code optimizations, even with fixed input data.

However, our article does not yet give information about the decidability of phase ordering or parameters space exploration if the performance prediction function does not require program execution. Simply because the answer depends on the nature of such function. If such function is too simple, then it is highly probable that the phase ordering becomes decidable but the experimental results would be weak (since the performance prediction model would be inaccurate). The problem of performance modeling then becomes the essential question. As far as we know, we did not find any model in the literature that has been formally validated by statistical fitting checks as explained in [18].

8. REFERENCES

[1] A. Cohen and S. Girbal and O. Temam . A Polyhedral Approach to Ease the Composition of Program Transformations. In *Proceedings of Euro-Par’04*, Aug. 2004.

[2] B. Aarts and M. Barreteau and F. Bodin and P. Brinkhaus and Z. Chamski and H.-P. Charles and C. Eisenbeis and J. R. Gurd and J. Hoogerbrugge and P. Hu and W. Jalby and P. M. W. Knijnenburg and M. F. P. O’Boyle and E. Rohou and R. Sakellariou and H. Schepers and A. Seznec and E. A. Stohr and M. Verhoeven and H. A. G. Wijshoff. OCEANS: Optimizing Compilers for Embedded Applications. In *Proceedings of EuroPar’97*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[3] C. Click and K. D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988.

[5] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[6] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5, 2003. Electronic journal : www.jilp.org.

[7] S. M. Freudenberger and J. C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Code Generation – Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 146–172, London, 1992. Springer-Verlag.

[8] M. Frigo. A Fast Fourier Transform Compiler. In *Proc. of Programing Language Design and Implementation*, 1999.

[9] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[10] J. P. Jones. Universal Diophantine Equation. *Journal of*

Symbolic Logic, 47(3):403–410, 1982.

[11] K. Kennedy, N. McIntosh, and K. McKinley. Static Performance Estimation in a Parallelizing Compiler. Technical Report CRPC-TR92204, Center for Research on Parallel Computation, Rice University, May 1992.

[12] L. Almagor and K. D. Cooper and A. Grosul and T. J. Harvey and S. W. Reeves and D. Subramanian and L. Torczon and T. Waterman. Finding effective compilation sequences. In *Proceeding of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, Washington, DC, June 2004. ACM.

[13] M. E. Wolf and D. E. Maydan and D.-K. Chen. Combining Loop Transformations Considering Caches and Scheduling. *International Journal of Parallel Programming*, 26(4):479–503, 1998.

[14] M. Pschel and J. M. F. Moura and J. Johnson and D. Padua and M. Veloso and B. Singer and J. Xiong and F. Franchetti and A. Gacic and Y. Voronenko and K. Chen and R. W. Johnson and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE special issue on Program Generation, Optimization and Adaptation*, (2):232–275, 2005.

[15] M. Zhao and B. R. Childers and M. L. Soffa. A Model-Based Framework: An Approach for Profit-driven Optimization. In *ACM SIGMICRO Int. Conference on Code Generation and Optimization (CGO'05)*, San Jose, California, Mar. 2005.

[16] W. Mangione-Smith, T.-P. Shih, S. Abraham, and E. Davidson. Approaching a Machine-Application Bound in Delivered Performance on Scientifique Code. In *Proceedings of the IEEE*, volume 81, pages 1166–1178, Aug. 1993.

[17] Y. Matiyasevich. Elimination of quantifiers from arithmetical formulas defining recursively enumerable sets. *Math. Comput. Simul.*, 67(1-2):125–133, 2004.

[18] R. Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., New York, 1991.

[19] R. Whaley and A. Petitet and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–25, 2001. ISSN 0167-8191.

[20] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioglu. On Optimal Parallelization of Arbitrary Loops. *Journal of Parallel and Distributed Computing*, 11:130–134, 1991.

[21] T. Alexander. Performance Prediction for Loop Restructuring Optimization. Master thesis, University of Carnegie Mellon. Physics/Computer Science Department, July 1993.

[22] T. L. Veldhuizen and A. Lumsdaine. Guaranteed Optimization: Proving Nullspace Properties of Compilers. In *9th International Symposium on Static Analysis (SAS 2002). Lecture Notes in Computer Science.*, volume 2477, pages 263–277. springer, 2002.

[23] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the Limits of Program Parallelism and its Smoothability. In Wen-mei Hwu, editor, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 10–19, Portland, OR, Dec. 1992. IEEE.

[24] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler Optimization-Space Exploration. *Journal of Instruction-Level Parallelism*, 7, Jan. 2005. Electronic journal : www.jilp.org.

[25] K.-Y. Wang. Precise Compile-Time Performance Prediction for Superscalar-Based Computers. *ACM SIGPLAN Notices*, 29(6):73–84, June 1994.

[26] D. Whitfield and M. L. Soffa. An Approach for Exploring Code-Improving Transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.