HABILITATION A DIRIGER DES RECHERCHES

de l'UNIVERSITÉ de VERSAILLES ST QUENTIN-EN-YVELINES

Spécialité : Informatique

présentée par

Denis BARTHOU

Contributions à l'optimisation de code et à la génération de bibliothèques hautes performances.

Soutenue le 18 Février 2008 devant le jury composé de :

François	Bodin	Rapporteur
David	Padua	Rapporteur
Sanjay	Rajopadhye	Rapporteur
Albert	Cohen	Examinateur
Michael	Gerndt	Examinateur
William	Jalby	Examinateur

Habilitation préparée à l'Université de Versailles - S^t Quentin
au sein du laboratoire Parallélisme Réseaux Systèmes Modélisation (PR*i*SM)

Résumé

Le nombre de transistors des processeurs, ainsi que leur fréquence, ont suivi la loi de Moore pendant plusieurs décennies, au prix d'une complexité croissante des architectures. La fin récente de l'accroissement en fréquence a notamment deux conséquences: le parallélisme est désormais un des seuls vecteurs de gain de performances, et la chaine de compilation ainsi que le système d'exploitation sont indispensables pour l'obtention automatique de ces performances. Dû à la complexité des mécanismes architecturaux difficiles à modéliser de façon réaliste, les compilateurs restent cependant loin de pouvoir générer automatiquement des applications hautes performances, même pour un seul core.

Le travail que nous présentons se focalise sur d'une part l'optimisation et la génération de bibliothèques hautes performances et leur réutilisation automatique dans un contexte applicatif, d'autre part, sur l'évaluation et la modélisation des performances afin de guider l'optimisation. Les résultats de ces travaux sont suivis de perspectives de recherche.

Abstract

The number of transistors as well as the frequency of processors have followed Moore's law for the past decades, at the expense of an increase in architecture complexity. As improvements in processor clock frequencies have levelled out, a major shift to parallelism is taking place. Parallelism is now the only way to enhance performance, and compilers and operating systems are essential to improve performance automatically. Due to the complexity of hardware mechanisms, hard to model, compilers are still far from being able to generate high performance application codes, even on a single core.

The work presented in this thesis focuses on two aspects of this issue: we first describe results obtained for the optimization and generation of high performance libraries and their automatic reuse in applications, and then describe performance evaluation techniques and tools to guide the optimization process. Results of this work is followed by research perspectives.

Contents

Chapter 1

Introduction

This is a great time for High Performance Computing, in particular in the compilation field.

For the last decades, the semi-conductor industry for non-embedded systems has increased processor performance by increasing clock frequency and adding a limited degree of parallelism (through small vectors and instruction level parallelism). Raising clock frequency has one overwhelming advantage: With no modification of the code, provided the programming ISA remains backward-compatible, an application runs faster just by upgrading the hardware. The price to pay is higher hardware complexity due to the slower evolution of frequencies for memories than for processors: A hierarchy of caches and a set of complex mechanisms (prefetches, out of order execution, speculation to name a few) are necessary to fill processor pipelines and sustain high performance. However the technology seems to have reached a limit due to the underlying physics: the increase in frequency is associated with CPU die shrinkage and at the atomic scale this leads to electrical leakage, responsible for excess heat generation and power consumption, important production costs,...

As improvements in processor clock frequencies have levelled out, a major shift to parallelism, in particular to many-core parallelism is taking place. The only way to increase performance is to enhance parallelism, placing several processor cores on each chip. In contrast to increases in clock frequency, performance improvement on parallel machines depends essentially on parallel compilers, libraries and operating system. This is because each core is no more powerful than a traditional processor; the performance improvements come from using more than one core at the same time, and are obtained through the efficient interaction between tasks partitioning computation and data. However, generating high performance codes still remains quite challenging and is usually more complex than running in parallel efficient sequential codes. From a compiler perspective, generation of an effective parallel application requires at least three main ingredients:

- expression or detection of the parallelism
- code optimization and generation
- realistic architecture and performance model.

We briefly discuss the main issues concerning these ingredients.

Parallelism detection and dependence analysis have been under study for many years. While automatic parallelization has not met its promises, the efforts dedicated to the automatic detection of parallelism have led to efficient dependence analyses working on regular or irregular codes, for scalar, arrays and pointers (with alias analysis). Dependence analyses can exhibit parallelism that corresponds to a data-flow formulation of the sequential program, such as SSA form for scalars for instance, or single assignment form for arrays. The degree of parallelism detected is however limited: by dependence abstractions (constant distances, affine relations, ...) and by the implicit data-flow parallelism of the underlying sequential algorithm. In particular, finding parallelism in expressions, based on semantic properties of the operators, requires rewriting techniques that are out of reach of dependence analysis alone. Data-flow or stream-based languages (Khan network languages, Stream-IT [145, 69], Lustre [75], ...) explicit the flow of values, naturally associated

with a partial execution order corresponding to parallelism. The limits are still the same, no semantics of the operators, dependences constrained by language, but writing applications with explicit data-flow is assumed to favor parallel codes. Parallel languages with explicit parallelism, such as openMP or MPI let the user design its application from an algorithmic point of view and at the same time explicit coarse-grain parallelism. Finally, libraries and domain-specific languages (DSL, such as Matlab or SPL [129]) abstract away parallelism in order to focus only on the semantics of the functions. The concrete implementation of a function may be parallel or not and a multi-versioned code is able to select the best code depending on execution context and semantic properties of the algorithms. All these approaches to parallelism expression are well-known and are usually combined altogether: for instance, explicitly parallel or data-flow languages are used to express task parallelism, automatic parallelization finds instruction level parallelism and vector code, and parallel libraries are assumed to be used adequately. As Amdahl's law predicts, once parallelism is expressed, the resulting performance comes from the overhead induced by parallel constructors, and by sequential code optimizations.

Automatic code optimization and generation is usually achieved through a sequence of code transformations. While the effects of each transformation is well-known and there are transformations proposed to take advantage of each hardware mechanism, compositions of transformations have effects that are not very well evaluated. First of all, the difficulty arises from the phase-ordering problem, deciding which sequence of optimization to apply. This leads to the situation where optimization sequences are driven, in state-of-the-art compilers by heuristics. While this heuristics achieve good results in general, they show their limits for performance demanding codes and are not appropriate for the generation of high performance applications. For this reason, compilers propose pragmas and flags for the user to guide the optimization process according to the application. The difficulty also comes from the fact that architectures make the generation of high performance codes harder: many hardware mechanisms necessary to fill processor pipelines are fully controlled by the hardware, and their implementation on specific architecture is generally opaque. With these conditions, the impact of code transformations are difficult to assess precisely.

The consequence of increasingly complex hardware and optimization is that many optimizations are no longer beneficial for all programs and all inputs. In order to decide whether an optimization is worth applying or not, there is a need for a cost/benefit evaluation. This evaluation can be achieved through a performance model, experimental measures, or both. For performance sensitive to input, multi-versioned programs can overcome this difficulty and switch from one version to the other according to input values. Computing the point from which to switch to another version also requires performance evaluation according to input values. State-of-the art compilers only resort to performance model and a limited set of experimental measures through profiling, due to a limited amount of compilation time. On the contrary, iterative compilation is a pragmatic approach based on experimental search for library and application performance tuning. This exploration is guided by a performance model, by algorithmic knowledge of the application and can be limited by a finite set of compiler flags, of tile sizes values, ... The inherent limitation of experimental search comes from the fact that executions for some input data sets are assumed to be representative of executions for any other input data. The best performing code selected is assumed to be among the best performing code for all possible input data. This assumption requires at least algorithmic knowledge and a good performance analysis.

This is one of the challenges for compilers in the many-core era: at a time where the compiler chain is essential for performance improvements, the legacy of Moore's law of the last decades has widen the gap between machine models and real machines, making harder the automatic generation of high performance codes. One answer would be to simplify the real architecture to the point where it is possible to have a performance model for it. A more realistic approach is to use an empirical search on possible optimizations, guided by a simplified model. Auto-tuning methods for library generation, such as ATLAS, Spiral, FFTW or Staple, have shown the benefits of such approach. They combine algorithmic knowledge, performance analysis with empirical search and generate high performance codes, closing the gap with hand-tuned code. Generalizing this approach to other algorithms and libraries is still an open issue. Recent papers (known as the Berkeley view [10]) propose to focus the development of auto-tuning library generators for 13 categories of computing/communication patterns and algorithms defined as dwarves, that are essential building blocks for performance intensive applications. The old principle in software engineering that good code should

be reused instead of being rewritten from scratch led to the design of many libraries, but the hypothesis associated with dwarves goes a step further. The goal is that all applications are written as compositions of patterns/algorithms coming from dwarves, and the application performance is driven by performance of dwarf functions/templates.

This habilitation thesis focuses on the problem of automatic code optimization and performance evaluation, in particular in an environment with high performance libraries. Optimizing applications with high performance libraries can be considered as a divide-and-conquer approach to the optimization problem: some parts of the application are optimized independently of the application context. Then the whole application, using these already optimized code fragments, is in turn optimized. The work described in this document tackles two issues:

- How to divide the optimization problem? So far, the choice of the program computation to implement from scratch and to take from a library is left to the developer. The decision is made from an algorithmic perspective, based on a limited choice of predefined algorithmic building blocks proposed by libraries and used by many applications. Obviously this approach does not scale well as soon as the number of functions proposed by libraries increases. The developer has to spend more time finding library functions that correspond to parts of the applications, and usually there are several possible solutions (a classical example is the computation of a convolution, written either directly or by a simpler computation in the frequency domain). This is not an issue as far as libraries are only considered for code reuse, but this is a problem when performance is concerned. For BLAS for instance, the library interface was designed by hand and counts a small number of functions. In contrast, the more recent library generator SPIRAL is able to generate highly tuned parallel code, using algorithmic knowledge, for any formula on a set of predefined operators. Deciding which part of an application correspond to a computation that SPIRAL can handle is tricky, to say the least. Moreover, SPIRAL has shown that the choice of the application decomposition into predefined operators is as important as the optimization of the operators themselves. Likewise, the dwarf approach is fundamentally based on a hierarchical decomposition (parallel templates/patterns using sequential components) and multiple decomposition will be possible for the same code. This evolution requires a more automatic approach to divide an application between libraries and developer code. We propose methods for automatic refactoring of code fragments using predefined library functions and methods for the definition of new library functions, through a hierarchical compilation of application codes. This is not opposed to the traditional of approach of libraries for code reuse. This only means that it may be necessary to change the decomposition proposed by the developer, for performance reasons.

- How to evaluate performance? The obvious way is to measure code performance for some input data but this only gives a very partial image of performance. While applications may have a small number of different input data sets (training sets for SPEC benchmarks for instance), libraries must exhibit performance for all possible valid input values. The other method for performance evaluation is based on static analysis of the code generated. This is very complex due to the large number of hardware mechanism that may impact performance and to the difficulty to have a valid and detailed model for them.

  We propose combinations of measures and model in two cases. In the context of optimization using many versions, we measure performance of particular functions, called *constant performance kernels*, and use performance model on applications that are decomposed into these kernels. The idea is to ensure good performance prediction in order to generate multi-versioned codes, based on only a few experimental measures. For a more general problem of performance tuning, we propose techniques and tools combining static performance analysis and dynamic performance evaluation. Comparisons of static/dynamic performance evaluations help in identifying performance problems and guide the optimization process.

The performance evaluation problem is fundamental in the optimization process. We focus in the rest of the chapter on performance model issues and the consequences for the generation of high performance codes. Some related works concerning performance modeling and possible solutions for the phase ordering problem

are first presented, then two theoretical formulations are proposed, one for the phase ordering problem, the other for the problem of finding the best parameters for a sequence of optimizations. The theorems resulting from these formulations draw hard limits on optimizations and in particular, on library generation, using empirical search. This corresponds to the context of the work presented in this document.

1.1 Performance Modeling and Phase Ordering Problem

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 [3, 55].
- *Static Algorithmic Models:* usually, such models are algorithmic analysis methods that try to predict a program performance [27, 105, 154, 143]. 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 [86, 150].

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 [82], the common mistake in statistical modeling is to trust a model simply because it plots a *similar* curve compared to the real plot. Indeed, this sort of experimental validation is not correct from the statistical science theory, and there exist formal statistical methods [82] 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.

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 [62]. Many other cases of inter-phase dependencies exist, but it is hard to analyze all the possible interactions [158]. Click and Cooper in [33] 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 [93], 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 [139], the authors target a similar objective as in [33]. 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 [168], 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. Here, 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 iteration, a new code optimization sequence is used [41, 150] 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 [34, 160]. As mentioned, such methods are devoted to regular high performance codes and only use loop transformation in the polyhedral model.

1.2 Towards a Model for Phase Ordering Problem

We present in this section a 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[1]. 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 1.3, 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.

In order to check that the execution time has reached some value T, we assume that there is a computable 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. We assume the program $\mathcal{P}$ always terminate. Let $t(\mathcal{P}, I)$ be the predicted execution time. 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.

Iterative Compilation

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.

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

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

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 1 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).

Pb. 1 (Iterative Compilation 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, the problem is to find 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?

Textually, this phase ordering problem tries to determine for each program and input whether there exists or not a compilation sequence s which results in an execution time lower than a bound T. It can be shown that if there is an algorithm that decides this problem then there is an algorithm that computes one sequence s such that $t(s(\mathcal{P}), I) < T$, provided that t always terminates.

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 1 when the cardinality of $\mathcal{M}^*$ is infinite while the set of distinct generated programs is finite remains an open problem.

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

Library Optimization

We provide here a variation on the 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 [129] and FFTW [63]. If the input is also part of the unknowns, the problem has the same difficulty.

Pb. 2 (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 2 are stronger than those of Problem 1: 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.

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

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 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.

Generative compilers making the assumption that sequences of best optimized codes are best optimized sequences fit the one-pass generative compiler description. For example, the SPIRAL project in [129] is a generative compiler. It performs a local optimization to each node. SPIRAL optimizes FFT formulae, from the formula level, by trying different decomposition of large FFTs. Instead of a program, SPIRAL starts from a formula, and the optimizations considered are decomposition rules. From a formula tree, SPIRAL recursively applies a set of program transformations at each node, starting from the leaves, generates C code, executes it and measures its performance. Using dynamic programming strategy[2], composition of best performing formulae are considered as best performing compositions.

As can be seen, finding a compilation sequence in generative compilation that produces the fastest program is a decidable problem. 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: 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 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).

Figure 1.1 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.

[2]The latest version of SPIRAL use more elaborate strategies, but still does no resort to exhaustive search/test.

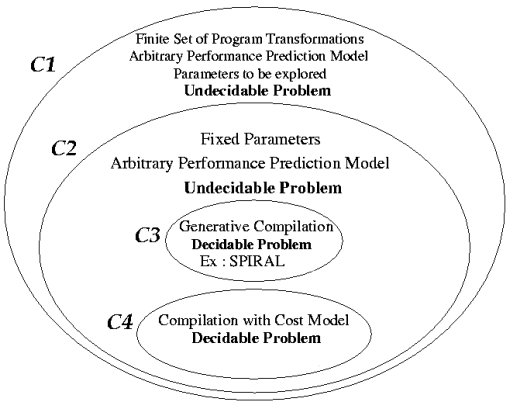


Figure 1.1: Classes of Phase-Ordering Problems

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

1.3 Towards a Model for Best Optimization Parameters

Nowadays, many compiler optimization methods are parameterized. 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.

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.

Polynomial Performance Model

In this chapter, 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 discuss about the choice of a polynomial model after the statement of the problem. We want to decide whether there are some parameters for the optimization modules that make the desired performance bound reachable:

Pb. 3 (Best-Parameters) *Let $\mathcal{M}$ be a finite set of program transformations and s a particular optimization sequence of $\mathcal{M}^*$. Let* 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 3 is decidable or not. Problem 3 considers polynomial functions, which are a family of usual performance models (arbitrary linear regression models for instance). Even a simple static model of complexity counting assignments evaluates usual algorithms with polynomials (n^3 for a straightforward implementation of square matrix-matrix multiply for instance). 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 theorem shows that if t is an arbitrary polynomial function, then Problem 3 is undecidable.

The following theorem states that Problem 3 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 number is fairly low compared to the number of optimizations found in state-of-the-art compilers (such as *gcc* or *icc* for instance). Now, if t is a polynomial and there are less than 9 parameters (the user has switched off most optimizations for instance): if there is only one parameter left, then the problem is decidable. For a number of parameters between 2 and 8, the problem is still open [107] and Matiyasevich conjectured it as undecidable.

Theorem 3 *The 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.*

Finite Parameter Space

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 OCEANS project [1], and a second one is the ATLAS framework [157].

The OCEANS project [1] optimizes a given program for a given data set by exploring all combinations of parameter values. Potentially, such value space is infinite. However, OCEANS 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 [150].

ATLAS [157] is another simplified case of the Best-Parameter problem. 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 levels of blocking controlled by three parameters, bounded by the cache size and a small number of loop interchanges possible (for locality). Exhaustive enumeration inside admissible values enable to find the best parameter value.

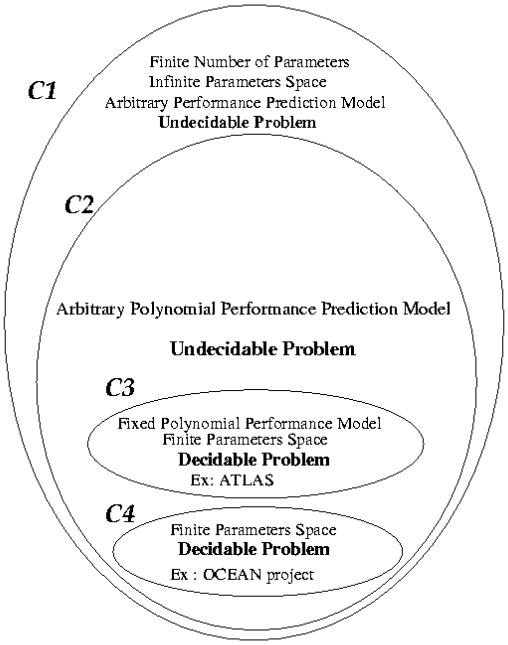


Figure 1.2: Classes of Best-Parameters Problems

Figure 1.2 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 OCEANS (class C_4).

1.4 Outline of the Document

The first two chapters are dedicated to optimization with and through high performance libraries. Chapter 2 presents automatic methods to detect then rewrite fragments of an application with calls to predefined high performance library functions and templates. Chapter 3 describes methods and performance model for a hierarchical approach to code optimization, defining libraries of kernels tuned for the optimization of a code. First applied to linear algebra library codes, the techniques are applied and adapted to more general applications.

Finally, Chapter 4 focuses on assembly code performance analysis and optimization. Based on a combination of static and dynamic analyses (instantiated in a tool, MAQAO), two approaches for optimization

are proposed, one of them building a set of binary templates instantiated and reused at run-time. Chapter 5 sums up the contributions of this work and proposes some perspectives.

Chapter 2

Refactoring with Performance Libraries

The subject of this chapter is to propose another way to improve application performance, by detecting automatically code fragments that can be replaced by high performance library function calls. All the work presented here, initiated in collaboration with Paul Feautrier and Xavier Redon [17] is fully described in Christophe Alias PhD thesis [4].

2.1 Introduction

The traditional approach in program optimization consists in applying successive transformations to the code in order to reach a high level of performance. Due to the large number of hardware mechanisms to trigger, the optimizations are complex and numerous, interactions between different optimizations are not fully comprehended and performance of the generated code is highly sensitive to the input data. As a consequence, there is a chance that compiler optimizations degrade performance, implying that performance tuning requires expert knowledge in architecture, optimizing techniques and application context, and a large amount of time in any case.

In order to reuse the effort and time spent to optimize similar code fragments, two alternatives are generally considered as a more reliable way to deliver performance for performance critical codes:

- High performance libraries: The assembly code of library functions is usually generated by hand, using architecture specific instructions, or by adaptative compilation (*e.g.* ATLAS [156], Spiral [129], FFTW [64]). The important compilation time is then balanced by the reusability of the libraries. In all cases, library functions can be considered as the building blocks, essential to get high performance on real codes. In general programming languages, code tuning is performed in the last stage of the development process. The selection of the library functions and the rewriting of the code falls under the responsibility of the user. The usual steps of this process are: find out code fragments and library functions that are semantically equivalent, replace these fragments by function calls with correct parameters, debug the application and finally evaluate performance. In the case of non-portable libraries, this time consuming process has to be reconducted for each target architecture. This is surprising how little the compiler helps the user in this tedious task. Compile-time optimizations neither change the partition between library and user code, nor cross library boundaries. This stresses furthermore the importance of this manual partitioning for performance.

- Domain specific languages: With domain-specific languages, algorithms are described in a more abstract and compact way than with traditional imperative programming languages. Expressing algorithms at a higher level of abstraction adds portability and improves programmer productivity for code writing and maintenance. All domain specific languages are not performance oriented, however

the higher the representation of the program, the more aggressive the compiler optimizations can be: for instance a generative approach can yield from a mathematical formula in SPIRAL [129] a finely tuned code for a particular architecture, exploring both algorithmic variations and traditional code optimizations, finding a large degree of parallelism. The same benefit appears also for code verification. However, this approach assumes that the user specifies his algorithm using a domain-specific language. Starting from an existing C or Fortran program and finding in it fragments that correspond to domain specific templates would be desirable but seems difficult to obtain.

Of course, the generation of these high performance building blocks (either libraries or basic operations for the DSL) is still an issue and it will be discussed in next chapters. Assuming there are high performance libraries available (or a DSL with its types and operations), developers starting from a code, written in C for instance, still have to decide manually where and when to call performance libraries. This transformation is called here code refactoring with libraries, as refactoring [60] stands for
"...restructuring an existing body of code, altering its internal structure without changing its external behavior."

This chapter explores methods that automatically find in a program the code fragments that correspond to known library functions. This enables a refactoring of the code, using high performance library function codes. To some extent, it is possible in some cases to rewrite a C code into a DSL program, based on the decomposition into library functions. Recognizing functions in a code and rewriting the code with these function calls in order to improve performance is a difficult task. As a matter of fact, this approach is not easier than applying transformations on a code in order to obtain high performance. Indeed it boils down to algorithm recognition, an old problem in computer science: Basically, one would like a compiler to automatically find that lines 10 to 23 are an implementation of a DFT for instance. A natural solution would be to search the code for patterns of known functions (matrix-matrix product, tensor product or direct sum for the DFT). Such a facility would enable many important techniques, that are out of reach for traditional optimization techniques:

- Program comprehension and reverse engineering: we can rewrite part of the code with a higher level language, enhancing code maintenance and portability.
- Program optimization: if we have the necessary items in our library or if we can use a generative approach, we may replace lines 10 to 23 by an automatically tuned version. We may even replace the relevant part of the code by a completely different implementation.
- Program verification: if we know that the program specification asks for a DFT and the analyzer does not find it, we may suspect an error.
- Hardware-software co-design: if we recognize in the source program a piece of code for which we have a hardware implementation (e.g. as a co-processor or an Intellectual Property component) we can remove the code and replace it by an activation of the hardware.

The approach described in this chapter relies on the results of a program dependence analysis and is threefold: (i) Find efficiently all code fragments of the applications that "may" be similar to some known functions implemented in high performance libraries. These code fragments are called *slices*. The linear complexity of this step is important since it analyzes all the application code and dependence analysis is a simple SSA. (ii) Compare the selected slices with library functions and keep only those that "must" be equivalent. Here, the dependence analysis required is an instance-wise reaching definition analysis (iii) Replace the code fragments selected by the appropriate function call in the program, or build a higher representation of the code in some DSL (we will consider this problem with SPIRAL).

The following sections follow this sequence. In the rest of the introduction, we propose a motivating example, showing all the steps required for the algorithm detection. We then provide some definitions for the equivalence, and introduce some required technical notions. Section 2.3 presents the fast detection of a potential pattern in the code, Section 2.4 describes the instantiation test and finally, we describe in Section 2.5 the conditions for which code substitution by function calls is possible and beneficial.

2.2 Program Equivalence

We focus on imperative programs written in C or Fortran. The libraries considered are template libraries (with the meaning of C++ templates). A template function, also called *pattern*, is a function depending on input variables and on some input functions (usually these functions are associated to the class of the variables given as input). Note that as a prerequisite for the detection step, each library function has to be described by a program. We do not assume that the analysis has access to the source of the library. Instead we assume the library designer provides a public version for each function. This program must have the same semantic as the optimized, private version but the algorithm used can be completely different. Moreover, we assume the library templates are static control programs [59].

Finding instances of a pattern in a program means finding all program slices *equivalent* to an instance of the pattern. But what does exactly mean "equivalent"? We will consider a weak version of semantic equivalence called *Herbrand-equivalence*. Instead of deciding whether two algorithms compute the same (mathematical) function, Herbrand-equivalence just states that they compute the same mathematical formula, syntactically (no associativity, no commutativity involved for instance). It means that if we consider the values computed by a code as terms depending on the inputs, an equivalent program computes exactly the same terms. In this way, Herbrand-equivalence can be considered as a true algorithmic equivalence. Even if Herbrand-equivalence is weaker than semantic equivalence, and seems to be easier to check, we have unfortunately proved that it is undecidable, even for the simple comparison of two programs (no template):

Theorem 4 *The problem of Herbrand-equivalence between two static control programs is undecidable.*

The proof is given in [17] and boils down to the tenth Hilbert Problem. It entails that the problem of matching, that is, finding all instances of a template in a code is also undecidable. Within this framework, the method presented is conservative: some of the fragments found are not semantically equivalent to the library codes, but none of the truly equivalent fragments is missed.

Obviously the equivalence is interesting only if a slice can be considered as an instantiation of the template even when they are not syntactically equal (otherwise simple pattern matching technique is sufficient). Herbrand equivalence can abstract away some variations between the slice and the template function. Program variations can arise from data structure variations (coming from scalar promotion, array expansion, structures instead of arrays,...), control variations (coming from loop fusion, unroll, skewing,...), organization variations (coming for permutation of program statements) or semantic variations (using associativity or commutativity for instance). Only semantic variations are not handled by the techniques proposed in this chapter.

The equivalence problem can be seen as a way to find sequences of transformations (control and data layout) where the objective is no longer to increase performance, but to involve a given high performance template.

2.3 Finding Slices similar to Template

The detection of library templates consists in localizing in a code the lines that possibly correspond to a given library function or template. In the case of a template, the code detected is a possible instance of the template. We propose an efficient method based on a symbolic execution of both program and template, following the def-use chains. The method symbolically executes both program and template slices simultaneously and compares the sequence of operators along these slices, abstracting away the number of iterations of the loops.

We assume that programs and templates are given in SSA form, a classical form in compilation that provides def-use chains. For the sake of clarity, we assume that each statement has one operator at most. Each edge of the program SSA-graph is labeled with its operator. Loops create cycles in this graph but we abstract away the number of iterations. The sequence of operators along a path is considered as a word and the graph can be considered as a finite automaton. The idea of the algorithm is to check whether the language of operators generated by some code fragment is included in the language of operators generated

by a library function. Intuitively, this ensures that the same sequence of operations can happen in the code and in the library function.

Consider the toy example given in figure 2.1. Statements assigning a constant such as T_1, P_1 or P_2 have no predecessors in the graph of def-use chains, they can be taken as a starting point for the inspection. Following the def-use chains, starting from P_1, the sequence $\xrightarrow{1} P_1 \xrightarrow{1/.} P_4 \xrightarrow{1+.} P_5 \xrightarrow{\exp} P_6$ is

(a) Template (b) Code

Figure 2.1: Example of Template and Program. The template corresponds to a slice of the program.

obtained. Likewise for the template a possible sequence of operators is:

$$\xrightarrow{1} T_1 \xrightarrow{X(.)} T_2 \xrightarrow{1+.} T_3 \xrightarrow{\exp} T_{STOP}.$$

Walking through both program and template, with the condition that for each transition, the operator must be the same, we obtain the sequence:

$$\xrightarrow{1} (T_1, P_1) \xrightarrow{1/.,X(.)=1/.} (T_2, P_4) \xrightarrow{1+.} (T_3, P_5) \xrightarrow{\exp} (T_{STOP}, P_6).$$

This provides the candidate slice $\{P_1, P_4, P_5, P_6\}$, that possibly corresponds to the template provided that $X(.) = 1/.$ (this condition appears on the transition). This condition is necessary for the sequence to be the same for both template and program. Note however that the method will not check the coherence between the values of template variables if they appear multiple times. Likewise, the number of iterations in loops or the branches chosen in conditionals are ignored. These important points will be checked during the exact instantiation test (see Section 2.4).

Because most operators have an arity greater than 2, word automata are not expressive enough in general. Our detection algorithm, detailed in [7, 8] uses a powerful extension of automata called tree-automata. A *tree automaton* is a tuple $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$, where Σ is a signature, Q the set of states, $Q_f \subset Q$ the set of final states, and Δ a set of transition rules of the type $f(q_1 \dots q_n) \longrightarrow q$, where $n \geq 0$, $f \in \Sigma$ and $q, q_1, \dots, q_n \in Q$. Tree automata were introduced by Doner [51, 52] and Thatcher and Wright [142] in the context of circuit verification. Most of usual operations on word automata (determinization, minimization, Cartesian product, ...) extend naturally to tree automata [36]. There is no major difference with the word automata: we associate a state to each assignment then we add transitions according to the dependences given by the ϕ-functions. X, as a wild-card can be unified to any other operator.

Consider the pattern and the program given in figure 2.1. For sake the of clarity, we have chosen a pattern and a program with unary operators which will lead to the word automata given in figure 2.2. But of course, our method can handle operators with any arity. The ϕ-functions can be seen as multiplexers selecting the last definition for a given value.

The idea is now to step simultaneously the two automata down to the pattern final state (in the Figure, beginning with `r4=...`) while the operators are equal. The two automata have as many entry points as constant leaves (0 and 1 here), and we have to start a comparison from each couple of leaves. The operations corresponds to the definition of the *Cartesian product* of the pattern and program automata. The detected slices can then be computed by collecting all program states along the paths from initial states to each state with a final pattern state.

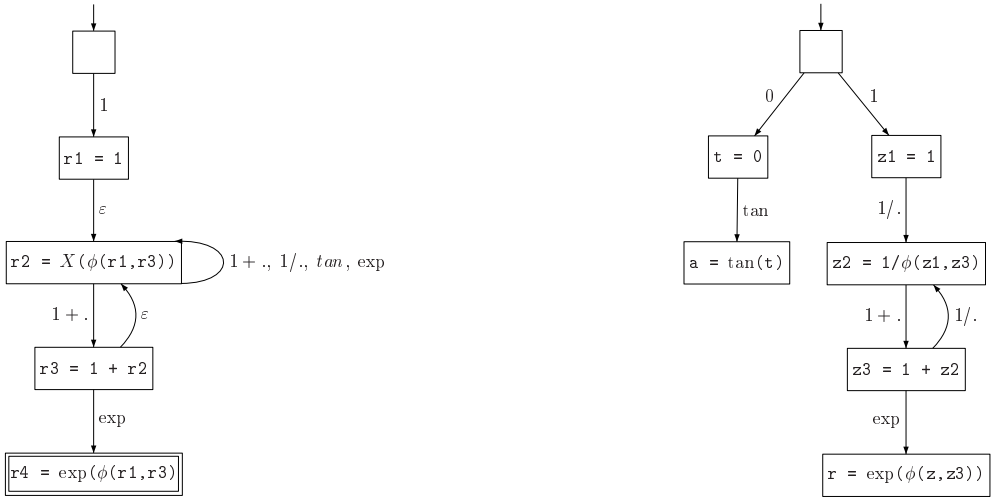


Figure 2.2: Example of Pattern automaton (left), and Program automaton (right)

This method is able to detect any template variation which does not involve the semantic properties of operators such as associativity, or commutativity. Particularly it handles any loop transformation and most control restructuring transformations. Moreover, our method is completely independent of data structure used, which allows the detection of a large amount of template variations in the program. Whether a slice detected is a real instantiation of the template is determined during the exact instantiation test (next section). In the worst case, the construction of the Cartesian product of the template and the program automata is computed in $O(T \times P)$ where T is the number of template statements and P is the number of program statements, i.e. the complexity is linear in the size of the program analyzed.

2.4 Exact Template Instantiation

The exact instantiation test consists in deciding whether a particular code is an instantiation of a given template. Templates considered are static control programs, and the code compared to is one of the slices extracted described in previous section. A template is considered as a program with some re-definable functions and inputs, as templates in C++ operating on parametric types.

For a program P, $P\downarrow$ denotes the term obtained by completely unfolding the computation of P. This term is well-defined for a program P with a schedule (no infinite loop). Likewise, for a template T with a schedule, the term $T\downarrow$ exists. These terms correspond to the symbolic term computed by the codes depending on the inputs. The matching problem is to find a substitution σ of the template variables such that $\sigma(T\downarrow)$ is syntactically equal to $P\downarrow$. Such a substitution, if it exists, is called a *unifier* of the matching problem between T and P. This problem depends clearly on the underlying algebra. It is clear, however, that equivalence in the Herbrand universe implies equivalence in all conforming algebras. We only consider in this chapter equivalence in the initial algebra (no semantics). In particular ϕ-functions are not given any interpretation during the resolution of the matching problem.

In order to give the intuition of the method, we study the case of the template and code in Figure 2.3 when $n = 2$. We can completely unfold the recurrence of the template, the output is:

$$X(X(Y[0], Y[1]) * 2, Y[2]) * 2$$

For the program in SSA form (Figure 2.3(c)), the output is:

$$\phi_{OUT}\left(0, \phi_{S_3}\left(\phi_{S_1}\left(0, S_3[1] * 2\right), \phi_{S_1}\left(0, S_3[1] * 2\right) + 1\right) * 2\right)$$

```
a = Y[0]
do i = 1, n
  a = X(a,Y[i])*2
enddo
OUT = a
```

(a) Template

```
MAX = I[0]
S = 0
do i = 1, n
  if (MAX < I[i]) then
    MAX = I[i]
    S = S+1
  endif
  S = S*2
enddo
OUT = s
```

(b) Slice to analyze

```
M0 = I[0]
S0 = 0
do i = 1, n
  S1 = φ(S0,S4)
  M1 = φ(M0,M3)
  if (M1 < I[i]) then
    M2 = I[i]
    S2 = S1+1
  endif
  S3 = φ(S1,S2)
  M3 = φ(M1,M2)
  S4 = S3*2
enddo
OUT = φ(S0,S4)
```

(c) SSA form of the slice

Figure 2.3: Example of template and slice to compare. X and Y are input function and array of the template.

with

$$S_3[1] = \phi_{S_3}\left(\phi_{S_1}\left(0, S_4[0]\right), \phi_{S_1}\left(0, S_4[0]\right) + 1\right)$$

Note that in the preceding expressions, $S_4[0]$ is not defined, its value will be denoted $\perp$.

The problem is to find out the possible values of X and Y such that the two expressions are the same, syntactically. Hence, the problem of matching is reduced to a semi-unification problem, between the terms of the template and of the program, where *unknowns* are template variables (here X and Y), and *closed variables* are program inputs (here $I[0]$, $I[1]$ and $I[2]$).

To unify two terms, we can apply Huet procedure [77]. Basically, we try to decompose the problem by applying the rule: $f(\vec{t}) \stackrel{?}{=} f(\vec{t'}) \rightarrow t_1 \stackrel{?}{=} t'_1 \wedge \ldots \wedge t_n \stackrel{?}{=} t'_n$ until terms of the form $X(\vec{t}) \stackrel{?}{=} f(\vec{t'})$ are obtained, where X is a template variable. Then we try to construct an unifier by trying:

- Projections: $X(\ldots x_k \ldots) = x_k$;
- Imitation: $X(\vec{x}) = f(X_1\vec{x} \ldots X_n\vec{x})$, with the X_i new function variables.

When a decomposition is not possible because the two head operators are different, then this is a failure, the program and template are not equivalent. When one of these operators is a ϕ-function then this function has to be computed in order to avoid the failure of the method. The value of a ϕ-function can only be one of its arguments. A dependence analysis may be enough to find out the argument defining the function. Otherwise, more complex data-flow analyses are necessary [59, 127, 108]. It may happen that a ϕ-function cannot be evaluated when the code containing it is outside the scope of the data-flow analysis. In this case, the ϕ-function remains uninterpreted and the unification fails, the template is then said non-equivalent to the program. This is a safe approximation.

On the example, we try to decompose the head operator in both expressions: for the code, this is ϕ_{OUT}, for the template this is $*$. Thus ϕ_{OUT} must be computed: as we have supposed that $n \geq 0$, a dependence analysis can show that $\phi_{OUT}(x, y) = y$. This is equivalent to prove that at least one iteration of the loop is executed. The head operator is $*$ now in both terms. By decomposition, we have to match $X(X(\ldots), Y[2]) \stackrel{?}{=} \phi_{S_3}(\phi_{S_1}(\ldots), \phi_{S_1}(\ldots))$. By applying the rule imitate, we define $X(x, y)$ as $\phi_{S_3}(X_1(x, y), X_2(x, y))$ where X_1 and X_2 are new template variables and the next step is a decomposition for ϕ_{S_3}. Note that here, there is no need to compute the exact value of ϕ_{S_3}. Finally, by an exhaustive application of the procedure, the solution $X(x, y) = \phi_{S_3}(\phi_{S_1}(0, x), \phi_{S_1}(0, x) + 1)$ and $Y[0] = \perp$ is found. The semantics of the ϕ-functions is:

$$\phi_{S_1}(x, y) = \textbf{ if } i > 1 \textbf{ then } y \textbf{ else } x, \phi_{S_3}(x, y) = \textbf{ if } M_1[i] < I[i] \textbf{ then } y \textbf{ else } x.$$

Therefore, the function X of the template of Figure 1.b is, in a more natural way:

$$X(x,y) = \left| \begin{array}{l} \textbf{if } M_1[i] < I[i] \\ \textbf{then } \left(\left| \begin{array}{l} \textbf{if } i > 1 \\ \textbf{then } x \\ \textbf{else } 0 \end{array} \right| \right) + 1 \\ \textbf{else } \left| \begin{array}{l} \textbf{if } i > 1 \\ \textbf{then } x \\ \textbf{else } 0 \end{array} \right| \end{array} \right.$$

Computation of M_1, used by X, can be added likewise to the definition of X.

In order to generalize this intuition for programs and templates with parametric loop bounds, the terms considered have a parametric length. Huet rewriting procedure is then no longer applicable as it is. The key idea is to build a finite automaton that compares the expressions computed by the program and the template and then to check if the states corresponding to failures in the unification procedure are reachable. The automaton proposed in [6] belongs to memory state automata[21] (MSA). Final states in the automaton express either failure of the unification, or success. In the later case, the unifier can be found by computing the reachability relation of the final state. All relations on the edges of the automaton are in Presburger arithmetics (thanks to the fact that the template is a static control program), and computing reachability relation boils down to computing a transitive closure of affine relations. This is what limits the method to a semi-algorithm, since the computation of transitive closures is undecidable. The construction of an MSA in order to check if the template matches the program was first proposed in [17], limited to templates without variable functions, generalized to templates in [6] and extended to templates containing loops in [4] with tree automata.

2.5 Rewriting Code with Library Calls

Given a template, applying the slice detection method on a program, followed by the instantiation tests defines the set of every code fragments matching the template, with the corresponding instantiation of the template functions. Now, it remains to transform these code fragments into library calls with their appropriate parameters (the inputs of the functions). Several difficulties arise for this code transformation, particularly if we want to improve performance:

- Legality [7]: replacing a code fragment by a semantically equivalent call to a library function does not always preserve the application semantics. Indeed, the library functions are assumed to be pure (no side effect, inputs are not modified), thus any variable modified in the code fragment should not be used outside of it, with the exception of the value returned by the library function .When this condition is not met, the transformation can be legal only by duplicating some amount of computation. This possibility has not been studied here, due to the potential performance loss induced.

- Trivial template instantiations: a dot product can be considered as a particular matrix-vector multiplication. It entails that all dot products will be also detected as matrix-vector multiplications. Worse, all scalar multiplications can be seen as dot products of size 1 vectors. These special instantiations of templates, getting rid of one dimension, should not be considered for refactoring.

- Overlapping instantiations: when considering different templates, the same code fragment can be part of two or more template instantiations. Moreover, a matrix-vector multiplication is usually written as several dot products. Choosing to replace all dot products by their library counterpart may prevent the more profitable transformation of matrix-vector operations with library functions. A cost model, discussed in the following, is required in this case.

- Data-layout Transformations Overhead: one of the strength of the method is to cope with some data layout transformations. In particular, array copies are no-ops for the recognition procedure. These

copies may be necessary in order to match the arrays of the program with the inputs of the library function. They have a cost that must be taken into account in order to decide whether it is worth or not to use the library function. Similarly to the previous item, a cost model is required.

- Aggregation of instantiations [8]: different consecutive code fragments may match the same template. Aggregating these fragments may make a bigger template instantiation (of the same pattern or of another one). A possible solution, discussed in the following, would be to create a hierarchy of templates, making possible the detection bottom-up of some templates from the detection of simpler ones.

The detection techniques we proposed are able to detect the occurrences of template with only one output statement. Figure 2.4 provides an example of matching, where the template is the daxpy function of BLAS 1. Our slicing method yields the candidates slices S_1 and S_2, but the candidate $S_1 \cup S_2$ where $a = 2$, $\vec{x} = [s(1), s(2), s(3), u]$ and $\vec{y} = [1, 1, 1, v]$ is missing since its outputs are shared by several statements. The same problem arises from (partially) unrolled parallel code, there are as many instantiations as the unrolling factor. Here, the solution would be to vectorize `s(i)` with `u` in order to build a bigger vector for daxpy . In

```
do i = 1,n
| y(i) = a*x(i) + y(i)
enddo
return y
```

```
S1  do i = 1,3
S1  | s(i) = 2*s(i) + 1
S1  enddo
S2  u = 2*u + v
```

Figure 2.4: Two detections of daxpy

general, aggregation induces a hierarchy between algorithms, and particularly between templates. Typically, a daxpy is an aggregation of several scalar daxpy , and a matrix-vector product is an aggregation of dot products. Figure 2.5 provides an aggregation hierarchy between some BLAS 1 and 2 functions. $A \rightarrow B$ means "B is an aggregation of A instances". We proposed in [8] a simple method to perform aggregation

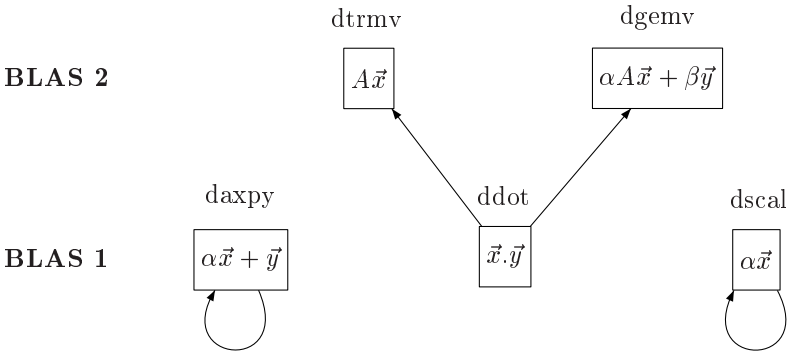


Figure 2.5: Aggregation hierarchy of BLAS 1 and 2 functions

based on a predefined hierarchy of templates.

In order to handle the overlapping of two or more detected functions, a benefit is associated to each substitution so that the combination of substitutions bringing the maximum benefit can be found by integer linear programming. The problem is similar to code instruction selection ([57, 54] to name a few), extended to expressions of parametric length with patterns including loops. A simple approach is to assume that the benefit corresponds to the gain (in cycles for instance) brought by the library function. As this gain depends on the number of loop iterations performed by the function, an approximation is to evaluate this gain based on the total number of iterations when the iteration domains are polyhedra [32]. In any case, the benefit of a substitution is expressed as a polynomial expression. A first approximation would be to select, by integer linear programming, the solution maximizing the benefit for asymptotically large iteration count, taking into account that templates overlapping are mutually exclusive.

Finally, our detection method handles variations due to data layout. It means that a dot product is detected on the following loop (from LINPACK):

```
do i = mp1,n-k,5
| dtemp = dtemp + a(k+1,k,i)*b(k+1,i) + a(k+1,k,i + 1)*b(k+1,i + 1) +
|         dx(i + 2)*b(k+1,i + 2) + a(k+1,k,i + 3)*b(k+1,i + 3) +
|         a(k+1,k,i + 4)*b(k+1,i + 4)
enddo
```

Rewriting this loop as a function call to a BLAS1 dotprod would require to copy elements of `a` and `dx` into the same array. The performance overhead added would be larger than any performance gain expected from the BLAS1 function. Finding an adequate performance model taking into account this kind of additional code (copy-in and out of intermediate data structures) is one of the major issues that need to be tackled in order to ensure that code refactoring with library functions improves performance.

2.6 Program variations detected

The efficiency of our approach directly depends on its capacity to recognize library functions in a source code. A common way to evaluate an algorithm recognition system is to provide the different kinds of pattern variations it can handle [159, 89, 106, 113]. We provide thereafter a detailed description of each variation. We also state whether our algorithm is able to detect them.

Organization variations Any permutation of independent statements and introduction of temporary variables. The following example provides an organization variation with legal permutations (LP), garbage code (GC) and temporaries (T):

```
s = a(0)
c = 0
do i = 1, n
| s = s + a(i)
| c = c + 1
enddo
return s + c
```

```
      s = a(0)
      c = 0
GC    garbage = 0
      do i = 1, n
LP    | c = c + 1
T     | temp = a(i)
      | do j = 1, p
GC    | | garbage = garbage + 1
      | enddo
      | s = s + temp
GC    | garbage = garbage + a(i)
      enddo
      OUTPUT = s + c
```

Our algorithm works on a def-use graph, which avoids the artificial precedence constraints due to the text representation of the program. This allows our algorithm to handle legal permutations and garbage code. Our method compares two by two the operators used in the template and the program, temporaries are therefore transparently coped with since they do not add operators.

Data structure variations The same computation with a different data structure. The following example gives a data structure variation with arrays and non-recursive structures:

```
s(0) = a(0)
do i = 1, 2*n
| s(i) = s(i-1) + a(i)
enddo
OUTPUT = s(2*n)
```

```
s.sum1 = a(0)
do i = 1, n
| s.sum1 = s.sum1 + a(i)
enddo
s.sum2 = a(n+1)
do i = n+2, 2*n
| s.sum2 = s.sum2 + a(i)
enddo
OUTPUT = s.sum1 + s.sum2
```

One of the important feature is the ability of the detection to cope with different representation of arrays. Transformations such as scalar promotion (transforming an array section into as many scalars) or array expansion (the reverse) are handled thanks to the aggregation step.

Control variations Any control transformation such as if-conversion, dead-code elimination and loop transformations as peeling, splitting, skewing, etc. The following example gives a control variation with a simple peeling:

```
s = a(0)
do i = 1,n
| s = s + a(i)
enddo
OUTPUT = s
```

```
s = a(0)
s = s + a(1)
do i = 2,n-1
| s = s + a(i)
enddo
s = s + a(n)
OUTPUT = s
```

Each of these variations provide an Herbrand-equivalent slice, which our algorithm is able to detect in a general way. But we are not able to detect non-Herbrand-equivalent variations, such as *semantics variations*, which uses semantic properties of operators such as associativity or commutativity. Nevertheless, experimental results given thereafter shows that our method finds a large amount of correct candidates.

2.7 Related Works

The detection of code matching library functions is related to the detection of slices, which consists in identifying all the statements contributing to a given computation. Program slicing was first introduced by Weiser [155] to help programmers debug their code. He defined a *slicing criterion* as a pair (p, V), where p is a program point and V a subset of program variables. A program slice for the criterion (p, V) is a subset of statements giving the same values to V as in the original program at point p. Weiser has shown that computing the minimal subset of statements that satisfies this requirement is undecidable [155]. However an approximation can be found by computing consecutive sets of statements according to data and control-flow dependences. Cimetile *et al.* [31] enhance the slice description with pre- and post-conditions for the detection of more specific slices. However their method requires user interaction in order to assert some invariants and has a high complexity, thus is not adapted to large applications.

Another approach proposed by Paul and Prakash [121] describes an extension of `grep` in order to find *program patterns* in source code. They use a pattern language with wild-cards on syntactic entities *e.g.* declaration, type, variable, function, expression, statement,... allowing to search for specific sequences and nested control structures. For example, here is pattern used to find the maximum in an array of integers in a loop:

```
{*
  @[while|dowhile|for] {*
    if($v_1[#] > $v_2) {*
      $v_2 = $v_1[#]
    *}
*}
```

The wild-card `{* *}` means a statement with an arbitrary nesting depth. Their algorithm has a $O(n^2)$ complexity with n the code size. This detection method has the same goal as ours; one of its drawbacks is that the same pattern cannot handle variations in control (loop unroll, tiling) or in data structures (array expansion, scalar promotion) whereas this is addressed in our framework.

Several approaches encode the knowledge of the functions to be identified in the form of programming plans. Top-down methods [89] use the knowledge of the goals the program is assumed to achieve and some heuristics to detect both the program slice and the library functions that can achieve these goals. Bottom-up methods [106, 159] start from statements and try to find the corresponding plans. Wills [159] represents

programs by a flow-graph, and patterns by grammar rules. The recognition is performed by parsing the program graph according to the grammar rules and has an exponential cost at worst. Metzger and Wen [113] have built a complete environment to recognize and replace algorithms. They first normalize both program and pattern abstract syntax tree by applying usual program transformations (if-conversion, loop-splitting, scalar expansion...). Then they consider all strongly connected components in the dependence graph, containing at least one `for` statement as candidate slices. Their method provides therefore a large number of candidate slices with many false detections, which is balanced by the low complexity of there equivalence test. Compared to the combination of the detection with the instantiation test we propose, they can handle fewer program variations (reuse of temporaries across loop iterations for instance is not handled) for a lower cost.

2.8 Experimental results

The implementation of the algorithm recognition system presented in this chapter has been developed by Christophe Alias in a tool, called `TeMa` [5] (for `Te`mplate `Ma`tcher). `TeMa` has been implemented in Objective Caml, and represents 10 kloc. `TeMa` uses the LLVM compiler infrastructure [96], which is based on `gcc` front-end, and is able to handle C programs. Additionally, a simple `fortran` parser has been written.

The detection and substitution of BLAS1 and BLAS2 functions in SPEC and PerfectClub benchmarks are detailed here. The benchmarks considered are `applu`, `swim`, `apsi`, `mgrid` from SPEC and `qcd` and `tis` from PerfectClub. The detection focuses on library functions from BLAS1 and BLAS2 and our pattern base is constituted of direct implementations of these functions form the mathematical description. As semantic variations are not handled by our method, different versions, based on semantic variations of library functions are added to our pattern base. For the detection, the method is in two passes, the first one detects candidate slices in linear time and the second checks that the candidate slices correspond to the library code. In average, 35.9% of the slices detected in the first pass do not match a library function after the second pass, 56.4% match but over vectors with a too small rank to obtain speed-up (such as matrices with only 1 line). The remaining candidates (7.7%) match a library function and can be substituted. Our algorithm seems to have discovered all of them, and particularly hidden candidates. Indeed, most slices found are interleaved with the source code, and deeply destructured.

The slices detected are substituted by Intel MKL BLAS routines. The speed-up results, comparing the codes before and after substitution, are shown in Figure 2.6. The machine used is an Itanium2 897Mhz and 2GB of RAM and the compiler used is `g95`. The average speed-up is 1.14. Several factors, such as the

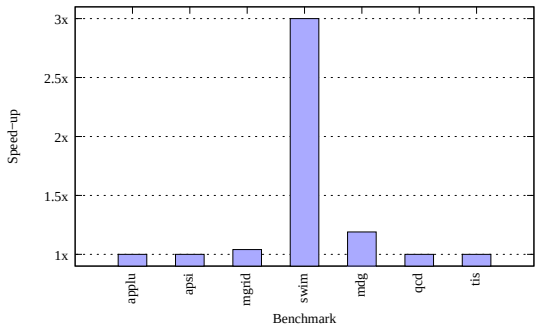


Figure 2.6: Speed-up on Itanium2 for SPEC and PerfectClub benchmarks, obtained by substituting automatically detected BLAS1/BLAS2 functions with MKL function calls.

number of detected slices and the weight of the slices that are substituted in the execution time, account for the speed ups presented here. For `swim`, the speed-up of 3 is explained by the detection and substitution of 6 daxpy functions that represent a large amount of the execution time. For `mgrid`, the substitution deals with 7 daxpy functions on small vectors, therefore leading to a small speed-up of 1.04. Several substitution

in `apsi` could not be performed due to dependences with other parts of the code. As for `applu`, `qcd` and `tis`, several BLAS functions were detected but they required memcopies, reducing the performance gain to a slow down.

BLAS3 functions can also be detected. 2 gauss functions for instance are detected in `applu`. However, intermediate results of the candidate slice were used outside of the slice, preventing substitution. For a more robust detection of complex functions, we believe that the aggregation technique and the hierarchy of templates need to be implemented in TeMa.

In addition, we show how `TeMa` can be used to recover an example of SPL program [163], representing computations handled by Spiral (For the moment, `TeMa` does not implement automatic SPL code generation). `TeMa` allows to recover SPL programs by recognizing and substituting SPL functions including matrix composition, direct sum and tensor product. Consider the following program, which is a naive implementation of the Walsh-Hadamard Transform (WHT):

```
   c(1) = f2
   do iter = 2,5
   |  rank = 2 ** (iter - 1)
⊗  |  do i = 0,1
⊗  |  |  do j = 0,1
⊗  |  |  |  do k = 0,rank-1
⊗  |  |  |  |  do l = 0,rank-1
⊗  |  |  |  |  |  c(iter,i*rank+k,j*rank+l) = f2(i,j)*c(iter-1,k,l)
⊗  |  |  |  |  enddo
⊗  |  |  |  enddo
⊗  |  |  enddo
⊗  |  enddo
   enddo
   wht = c(5)
```

`TeMa` detects the slice marked by ⊗ as a tensor product between `f2` and `c(iter - 1)`, and substitutes it in the following manner:

```
c(1) = f2
do iter = 2,5
|  rank = 2 ** (iter - 1)
|  c(iter) = f2 ⊗ c(iter-1)
enddo
wht = c(5)
```

Applying by hand our preliminary SPL code generation method, we finally obtain the following SPL program:

```
(tensor (F 2) (tensor (F 2) (tensor (F 2) (tensor (F 2) (F 2)))))
```

Even if the SPL generation is not yet implemented, the rewriting of the program with high-level functions increases readability, making `TeMa` a promising tool to improve program comprehension and help programmers in the tedious task of software maintenance.

2.9 Conclusion

This chapter proposes another approach to code optimization. Instead of relying on compiler optimizations, an alternative road to performance is to use high performance libraries for hot code fragments, whenever it is possible. Based on this essential notion of code reuse in software engineering, we propose a method that automatically finds the code fragments of an application that can be replaced by high-performance library calls, and rewrites the application accordingly.

More precisely, we described a new method to find template instances in large applications. The templates considered correspond to C++ templates, they must be static control programs, and must correspond to public code versions of library functions (the assembly code of the library is not directly used for comparison). There is no restriction concerning the application itself or the functions instantiated in the templates. The technique proposed is able to cope with many program variations, including all usual loop transformations, use of temporaries, permutation of instructions and many data structure variations (scalar promotion, array expansion, ...). The technique is therefore able to detect known templates that would be difficult to find manually. The algorithms have been implemented in a tool, Tema [5], based on LLVM compiler [96].

The advantages of code refactoring with high performance libraries are numerous:

- It first increases the productivity of the developer by performing automatically the tedious task of rewriting the code with high performance library functions. Even if an application is already written with library calls, our method is able to find all possible rewritings, possibly finding better versions. This answers to one of the issues expressed by Kennedy in [85] concerning the difficulty to do inter-procedural optimization of library functions. Indeed, it is possible to replace any function call by more specialized ones, according to the context, or by aggregating function calls into one function.

- Replacing code by a call to a known function is a gain of abstraction. This enables the use of DSL functions, and opens the door to algorithmic optimizations, using expert knowledge of a domain that would be out of reach from C programs. We have shown that it is in particular possible to find Σ-SPL expressions corresponding to a C program.

- Template libraries, as opposed to standard non-template libraries, widen the range of codes that can match. High performance libraries using templates (for instance STAPLE [147]) optimize the skeleton of the program (generally by parallelization) and let the user define the details of the program.

This work also shows the limits of this approach. The first limit is theoretical, since the template matching technique can only be a semi-algorithm. Independently of this hard limit, there are other issues, concerning the essential assumptions of the approach:

- We assumed that the application could be rewritten with known library functions. However, many applications cannot be rewritten with library calls, because there is no library function corresponding to the computation they perform. The main question is then how to generate libraries for these applications? We propose an approach to answer to this question in the following chapter.

- We assumed that replacing user code by high performance library functions improves performance. This is usually a safe bet. However, one of the strength of method proposed is to recognize functions even if there are some data layout transformations. It means that data layout transformations, such as copies, transpositions, are no-ops for the detection, unlike for pattern-matching techniques. When substituting code fragments by detected functions, these copies have to be added and their overhead is sometimes important. Therefore a performance model is required in order to ensure that replacing the original code with a library call improves performance. We tackle this issue in the following chapter.

Finally, there are many extensions possible to this work. In particular, a promising extension would be to handle semantic properties of operators. Actually, there are two possibilities for this: either this is handled on the program, before the detection of library functions. Or this is handled once some functions have been recognized in the code. The semantics is then between these functions, on a higher level of abstraction. Transformations using algorithmic knowledge and semantical properties are performed for instance on DSL such as Σ-SPL in Spiral.

Chapter 3

Library Generation with Hierarchical Compilation

Template recognition as presented in the previous chapter assumes performance critical parts of an application can be written with high performance library functions. Libraries, as usually considered, are efficient implementation of well-known algorithms in a specific area, that can be reused in multiple contexts. If we consider libraries as a way to optimize codes, then there are two main issues:

- How to generate high performance libraries for known algorithms ?
- How to create libraries for applications that do not correspond to existing libraries ?

This chapter tackles the first issue for linear algebra codes, and proposes some solutions for the second problem. In both cases, only mono-threaded applications are considered. Parallelism is limited to Instruction Level Parallelism and Vector operations.

Most of the results presented here are part of Sebastien Donadio PhD thesis [49].

3.1 Introduction

The increasing complexity of hardware features incorporated in modern processors makes high performance code generation very challenging. One of the key difficulties in the code optimization process is that several issues have to be simultaneously addressed: for example maximizing instruction level parallelism (ILP) and optimizing data reuse across multilevel memory hierarchies. Moreover, very often, a code transformation will be beneficial to one aspect while it will be detrimental to the other one. The whole problem worsens because the issues are tackled by different levels of the compiler chain: most of the ILP is optimized by the backend while data locality optimization is performed at a higher level. The result is that generating automatically (with a compiler) high performance libraries is still an issue, and most vendor libraries, at least for monocore architectures, have been generated by hand-code tuning.

An emblematic example of these problems is the simple dense matrix multiply operation. Although the code is fairly simple, none of the recent production compilers is really able to generate performance close to hand-coded routines. Whaley *et al.* [157] have developed a specialized and iterative code generator (ATLAS). ATLAS explores by empirical search different tiling parameters and scheduling optimizations. ATLAS generated libraries outperform codes produced by most of the compilers. Recently, the cost of the iterative compilation in ATLAS has been reduced by replacing the iterative search by a cost model, while still generating codes with nearly the same performance [165]. But even with these recent improvements, vendor [79, 115] or hand-tuned BLAS3 [70] still outperforms ATLAS generated codes. So what are ATLAS and compilers still missing in order to reach this level of performance?

The histograms in both Figure 3.1 and Figure 3.2 compare the performance of two versions of a square matrix multiply primitive (DGEMM): one is generated by ATLAS (grey bars) and the other one is from

the Intel MKL library (white bars). The target machine is an Itanium 2 running at 1.575 Ghz with a peak performance of 6.3 GFlops. MKL version clearly outperforms ATLAS version and gets performance numbers very close to peak. Trying to understand the performance gap, we measured L2 and L3 misses for each code. The results in Figure 3.1 (resp. Figure 3.2) represent the average number of bytes fetched outside of L2 (resp. L3) per FMA (Fused Multiply Add). These metrics are simply computed according to the following formula: $(128 \times number\ of\ L2\ misses)/number\ of\ FMA$ (resp. $(128 \times number\ of\ L3\ misses)/number\ of\ FMA$). Surprisingly enough, MKL is performing on average 3 times more L2 misses than ATLAS and between 1.5 and 2 times more L3 misses (for matrices larger than 900×900). We checked that the number of prefetch instructions is similar in both cases and we also looked at the impact of TLB misses: again in both cases the impact is negligible meaning that both ATLAS and MKL have done an excellent job at minimizing TLB misses. It should be noted that although the stress imposed by MKL routines on L2 bandwidth (resp. on memory bandwidth) is much higher, this is still below the sustained bandwidth achievable by L2 cache: 24 GB/s (resp. by memory system: 6.4 GB/s). Similar experiments performed on Pentium 4 lead to similar observations.

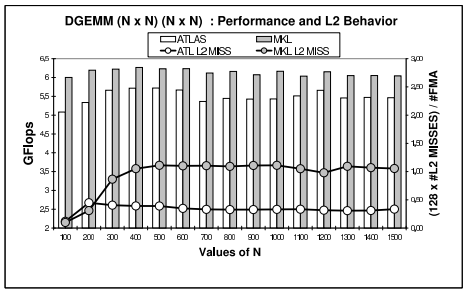


Figure 3.1: DGEMM performance and L2 behavior for ATLAS and MKL on Itanium 2

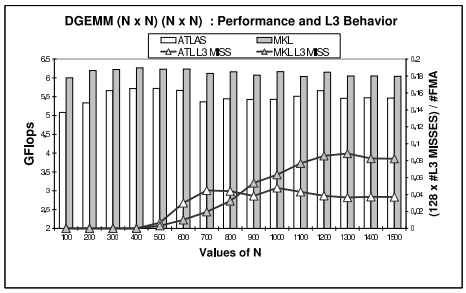


Figure 3.2: DGEMM performance and L3 behavior for ATLAS and MKL on Itanium 2

What conclusions should be drawn from this example?

- Minimizing L2, L3 and TLB misses should not be the only goals;
- Furthermore, increasing L2, L3 and TLB miss rate (but still staying under the hardware bounds) can be the right path to reach peak performance;
- The real key to achieve peak performance is to achieve the right tradeoffs between ILP optimization and data locality optimization.

In this chapter, we propose an automatic method to close this performance gap. The starting point is to decouple the two issues: locality optimization and ILP optimization. Tiling is first performed to produce a tile code operating on subarrays that fit in cache. Tiling constraints, ensuring that the subarrays accessed within the tile fit in cache, do not define a single tile size but rather a set of tile sizes. We use this degree of freedom to search for the best performing tile code and choose the tile size according to the constraints on this code. Then, to optimize the tile code, we use a bottom up approach combined with systematic exploration. In general, the multiply-nested loop structure of the tile code will still be too complex to be correctly optimized by a compiler even if the operands are in cache. Therefore, from the multiply-nested loop, we systematically generate several *kernels*, using interchange, strip mining, and partial unrolling. These kernels have a simpler control structure (1 or 2 dimensional loops), the loop body containing several statements resulting from unrolling surrounding loops. Additionally, to simplify the compiler task, loop trip counts are set to constants, and multidimensional array accesses are simplified. The main constraint on these kernels is to be simple enough so that a decent compiler can generate high performance code. Then, the performance of all of these kernels is systematically measured. From these kernels, different variants of the original tile

code can be easily rebuilt. And finally, taking into account tile size constraints, a specific version of the tile code is selected and the whole code is produced.

The proposed approach is demonstrated on linear algebra codes (BLAS, convolution, a LAPACK function) and on several applications. Unlike ATLAS, we did not *a priori* select a given code that is further tuned. On the contrary, we consider a large number of variants, which are automatically produced. Each of these variants correspond to the application of a given set of transformations/optimizations to the original tile code. Generation and exploration of these optimization sequences and their parameters are achieved with a meta-compilation language, X-language.

The main contributions of the proposed approach are:

- Automate the process of generating high performance code optimizing simultaneously ILP and data locality. The main contribution here is that our approach allows to find the best tradeoff between these two optimization targets
- Achieve performance similar to hand coded routines
- Rely on flexibility (different versions) of the code generated to match varying data locality properties. For example, the real difficulty is not generating a matrix multiply code achieving high performance on square matrices, but a general matrix multiply code that will obtain high performance for arbitrary rectangular shaped matrices (where locality properties on each array can be very different)
- Reduce the cost of the optimization phase. Most of the search phase and experimentation phase are done on the kernels (which are mostly one dimensional loops) and not on the whole code

It should be noted that our results are fundamentally different from those obtained by Chen *et al.*[30] that are essentially focused on data locality optimization across all of the memory hierarchy levels.

3.2 Representing Multiple Program Versions

Due to the complexity of the performance model for current architectures, as shown in Chapter 1, the problem of finding a sequence of transformations leading to a best performing program is untractable. Moreover, for the same reason, building a simplified performance model extrapolating a few performance measures does not in theory help much in delivering best performing codes. In this section, we focus on the exploration of multiple program versions with different optimizing parameters, for the same input.

Exploration of the optimization space for a program needs to be limited to a relatively small set of optimization sequences and parameter values. As all possible optimizations are not explored, one best performing code can only be reached if the subset explored contains it. The different approaches proposed in the literature to tackle this issue are:

- Explore a very large optimization space, using an incomplete performance model or a machine learning or genetic technique, or both.
- Explore exhaustively an optimization space in order to build some performance model.
- Assuming the developer has some *a priori* knowledge of the optimization space, depending on the application, the developer could describe in an appropriate language the optimization space to explore.

We present briefly in this section a new language to describe optimization sets.

In order to generate multiple versions of a code, we resort to a two-stage compilation framework. Source-to-source transformations are expressed in a meta-compilation language, called X-language [50], and are applied on the code after the first compilation stage. The second stage corresponds to a usual compilation phase. X-language is a language of pragmas, specifying sequences of transformations and range of parameters.

The main features of X-languages are:

1. Elementary transformations: The first feature that comes to mind are constructs to generate multiple versions of a statement by applying elementary transformations to a statement. Elementary transformations are widely used transformations that cannot be conveniently cast in terms of other, simpler transformations. For program optimization, the target of the transformations are usually compound statements and the transformation typically manipulate the the order of execution and the control structure of the components. Loop transformations include unrolling, interchanging, strip-mining, fusion, fission, and scalar replacement. Many elementary transformations require input parameters, such as the degree of unrolling, tile size, and locations where the loop is to be split in the case of fission. Multiple versions of the initial statement are obtained by varying the values of these parameters. For instance, applying a partial unroll on loop `i` with unrolling factors among the values of 1, 2 and 4 is specified with the following pragmas:

   ```
   #pragma xlang parameter UF [1,2,4]
   #pragma xlang transform unroll(i,UF)
   ```

   where the first pragma defines the possible values of `UF`, the parameter corresponding to the unrolling factor, and the second applies the unrolling transformation on the loop. ATLAS makes use transformations that are tiling, unrolling, and loop scheduling; FFTW makes use of loop scheduling (software pipelining); and SPIRAL applies loop unrolling for instance.

2. Compositionality: Usually the best version of a code is the result of the application of several elementary transformations. Thus, for example, ATLAS applies loop interchange, tiling, unrolling and basic block scheduling to the triply nested matrix-matrix multiplication loop during its empirical search for an optimal form of the loop. Therefore, our language allows the application of multiple transformations to a single code. Elementary transformations are seen as functions taking a code and returning a code. For instance, application of an interchange of loops `i` and `j`, followed by a partial unroll of `j` by a factor of 2 is described by:

   ```
   #pragma xlang transform inter(i,j)
   #pragma xlang transform unroll(j,2)
   ```

   The composition of elementary transformations builds a new transformation of the same type. Additionally, the transformations have other input parameters, such as unrolling factor.

3. Extensibility: For composite transformations, it is convenient to encapsulate them into new transformations and to avoid having to rewrite sequences of transformations that are applied more than once. This extension mechanism also enables the user to add new transformations that cannot be represented as a composition of elementary transformations. In particular, programmers should be able to generate application-dependent transformations that take into account the semantics of the computation. All elementary transformations and further extensions of X-language are defined in Prolog, a natural language to define manipulations of terms. This extension mechanism makes the definition of new transformations very easy and concise (just a few lines of Prolog are required to define classic transformations). For instance, fission of loops can be defined by:

   ```
   fission(I,for(I,LB,UB,S,(B1;B2)),
            (for(I,LB,UB,S,B1);for(I,LB,UB,S,B2))).
   ```

   The term fission has three argument: the first is the name of the loop to fission, the second is the loop before fission and the third is the loop after fission.

4. Second-order transformations and parameter ranges: This mechanism controls application of transformations, exploring transformations to apply on the code. The exploration can be either on different transformation sequences (in which case transformations take other transformation as parameters) or

on different parameter values (defining ranges). Search strategies can be defined likewise with more complex second order transformations. Actually, there is only an exhaustive search implemented in X-language.

With the help of X-language, we have shown that it is possible to mimic the exploration achieved in ATLAS (based on experimental search, not unleashed), obtaining comparable performance results for DGEMM. The current implementation of X-language is based on a C99 front-end parser, tiny C compiler [124] and relies on a Prolog engine for source-to-source transformations.

The main advantage of X-language is that the user can apply very precisely source-to-source transformations and define its exploration space in a practical manner. The initial program is not modified and can still be compiled by a production compiler, ignoring all pragmas. Checking the validity of the resulting transformations is done after the application of all transformations, between the initial code and the optimized version. However, long optimization sequences are tedious to write with pragmas. For instance, optimizing DGEMM from a naive version in order to obtain ATLAS performance requires a dozen of consecutive transformation pragmas. Moreover, transformations may generate multiple loops (tail codes for instance) that have the same loop counter. In order to be able to identify each loop individually, this requires special naming pragmas before each loop, and naming convention for newly created loops, obfuscating quickly X-language directives.

The following section defines a more automatic method, still based on a user-defined exploration space, but requiring far less pragmas and generating codes with higher performance.

3.3 Hierarchical Kernel Decomposition

The objectives of the decomposition are to generate high performance codes, with an empirical search among solutions trading off ILP and cache usage. As this optimization space is large, we need a way to decrease the combinatorial complexity: performance measures are only made on small code fragments, called kernels. These kernels partition the computation of the function to optimize and can be executed independently of the surrounding function, hence the term of hierarchical kernel decomposition.

Principle of the decomposition

As highlighted by Chapter 1, the performance of a code (even a simple one) cannot be easily predicted, and a few performance measures for some input values does not help to predict performance for other input values. To avoid this issue, we consider here only kernels with inputs that have no impact on performance. The following definition gives the conditions on the considered kernels:

Definition 1 (Constant Performance Kernel) *A constant performance kernel is a static control code, with numerical loop bounds and strides, constant array sizes, and for all arrays, the starting address has a constant offset w.r.t. the first address of a page.*

The remaining inputs (values of the array elements and some scalars) have no real impact on performance. Provided input data is in the same configuration in memory, two different executions of the same kernel, with different inputs, will take the same number of cycles. This is one of the key principles of hierarchical decomposition: we generate multiple versions of codes using different constant performance kernels, and kernels are associated to one performance figure if its dataset is in cache.

Taking into account these constraints on the kernels to generate, the main steps of the decomposition are:

1. Loop Tiling: The main goal of loop tiling[161, 35, 92, 91] is to reduce memory traffic by enhancing data reuse. Upon entry in the tile, data layout (the array organization) is restructured in particular to reduce TLB misses. Indeed, all copies are hoisted at the entry/exit of the tile, to generate contiguous array sections which will minimize cache interferences and TLB misses. At this stage, the only constraints imposed on tile sizes is that they should be chosen such that the working set used in the tile fits into

the cache. In general, the exact evaluation of the working set can be done but is fairly complex [32]. For our purpose, in most cases a simple method based on rectangular array subsections (derived from the tile sizes) is sufficient.

We do not limit our search to square tiles. While using general rectangular tiles will increase the number of parameters to deal with, it provides much more degrees of freedom. First, it allows to cope more efficiently with the case where the original loop iteration space is rectangular (allowing for example to deal with the general rectangular matrix multiply problem). Second, it allows even to deal efficiently with more arbitrarily shaped iteration space such as triangular ones.

2. Loop transformations: The transformation sequences to explore are described by X-language pragmas. For instance, for BLAS code, we considered only loop interchange, partial unroll, and loop fusion. The goal of these transformations is to increase the parallelism inside some loops (by unrolling or fusion), to give opportunities for higher instruction parallelism and vectorization. From the multiple versions generated, we extract the constant performance kernels.

3. Data layout transformations: The complexity of the constant performance kernels generated depends highly on the number of loops considered for the tiling. Kernels with only one loop (1D kernels) have several advantages: compilers usually generate good quality codes for single loops, and the experimental step takes linear time with respect to the number of experiments performed. Data-layout transformations are applied to simplify array structures accessed by a kernel and generate again multiple versions of kernels. The optimization is focused on:

   - locality optimization: higher locality and regular strides give more opportunities for the compiler to generate high performance memory accesses (better prefetch, fewer instructions to describe the address stream, vectorization,...)
   - register usage optimization.

   These are standard scalar promotion/blocking techniques. The first goal can be reached by transforming the arrays so that they contain only the working set accessed by the kernel. This can be achieved by using well-known techniques[97, 15, 146]. Our implementation is based on scalar promotion techniques, and all arrays are resized to the size of kernel loops. Array dimensions that are only indexed by constants or loop counters not present in the kernel are removed (and arrays are renamed correspondingly). This may require the use of memory copy operations.

The quality of the final code depends on the capacity of the compiler to take advantage of the parallelism expressed in the constant performance kernel. In particular, the compiler has to perform the following operations appropriately:

- Dependence analysis: failure to detect independence of statements degrades schedule latencies, ILP and impacts many other optimizations.
- Register allocation. Depending upon the architecture, this is more or less critical and failure to correctly allocate registers introduce dependencies, impacting latencies. Source-to-source transformations can help the compiler by using single assignment form codes.
- Vectorization: this can be memory access vectorization (Itanium, Pentium for instance) or computation vectorization (Pentium with SSE instruction set). Some code generators rely on pattern matching in order to decide whether or not to perform vectorization. Enumerating different versions of kernels helps in finding the appropriate code that can be matched by these rules.
- Constant propagation and use of static information: the compiler can take advantage for instance of the loop bounds values (which are explicitly provided) for the computation of the prefetch distance. Failure to do this means that the compiler optimizes in the same manner loops of different sizes.

In the following, we illustrate this technique on the matrix multiply code.

Example on DGEMM

Decomposition can be applied successfully on different algebra codes. We focus here on the well-known DGEMM. In Figure 3.3, we present the code of a tile of a matrix multiplication (also called mini-MMM). The code is annotated with X-language pragmas specifying the transformation sequences to generate multiple versions.

```
#pragma xlang begin
  for (i=0; i<NI; i++)
    for (j=0; j<NJ; j++)
      for (k=0; k<NK; k++)
        z[i][j] += x[i][k]*y[k][j];
#pragma xlang parameter Ui [1,2,4]
#pragma xlang parameter Uj [1,2,4]
#pragma xlang parameter Uk [1,2,4]
#pragma xlang transform any(inter)
#pragma xlang transform unrolljam(i,Ui)
#pragma xlang transform unrolljam(j,Uj)
#pragma xlang transform unrolljam(k,Uk)
#pragma xlang parameter NI [100,200,400,600,800]
#pragma xlang parameter NJ [100,200,400,600,800]
#pragma xlang parameter NK [100,200,400,600,800]
#pragma xlang transform decompose(1)
#pragma xlang end
```

Figure 3.3: Mini-MMM code with its annotated X-language transformations. `any(inter)` generates all versions due to possible loop interchanges. `unrolljam` performs unroll and jam on the loop with the specified iteration counter. All `parameter` pragmas describe the set of values to explore. `decompose` generates in separate files all the resulting different 1D constant performance kernels.

Figure 3.4 presents three mini-MMM optimizations achieved by our method. The first column gives the optimization sequence used and the resulting code is in the second column. From these codes, the kernels in the third column are generated by tiling of the innermost loop and scalar promotion. Scalar promotion creates new arrays. For the first example, the mapping between array elements of the first optimized version and array elements used in its 1D kernel is the following: `c1=c[i]`, `a1=a[i][k]`, `b1=b[k]`, `c2=c[i+1]` and `a2=a[i+1]`, where all arrays of the kernel have `ni` elements. In C, this implies that there is no need for copy at all. For the second optimized mini-MMM of Figure 3.4, the mapping is: `c1=c[i][j]`, `a1=a[i]`, `b1[.]=b[.][j]`, `c2=c[i][j+1]`, `b2[.]=b[.][j+1]`, etc. Note that array `b` has to be transposed before entering this kernel. For the third optimized mini-MMM, the mapping is: `c1[.]=c[.][j]`, `a1[.]=a[.][k]`, `b1=b[k][j]`, `c2[.]=c[.][i+1]` and `b2=b[k][j+1]`. This means that arrays `a` and `c` have to be transposed.

Concerning the mini-MMM code for DGEMM, enumerating all optimizations and generating all kernels leads to 5 different kernels, 4 of them are presented in Figures 3.5, 3.6, 3.7, 3.8. The remaining 3D kernel is the DGEMM itself. The values k, l correspond to the unrolling factor of the surrounding loops. For instance, `daxpy k,l` corresponds to daxpys accumulating in k different vectors, l daxpys sharing the same destination vector. All `ni`, `nj` are constant values. Optimizations such as prefetching may be influenced by the actual value of the bound and loop overheads and pipelines with large MAKESPAN or large unrolling factors can take advantage of larger iteration counts. The span of the loop bound sampling is user-defined through X-language `parameter` pragmas. As for array starting addresses, all possible offsets are tested (but this could be specified by pragmas). Indeed, the code generated may be unstable with respect to the alignment of the array starting addresses and important performance gains can be obtained by finding the best alignment[83].

Optimization	Optimized mini-mmm		1D Kernel

interchange j,k; partial unroll i.	`for (i = 0; i < NI ; i+=2)`
	`  for (k = 0; k < NK ; k++)`
	`  for (j = 0; j < NJ ; j++)`
	`  c[i][j] += a[i][k] * b[k][j];`
	`  c[i+1][j] += a[i+1][k] * b[k][j];`
		⇒	`for (i = 0; i < ni ; i++)`
			`  c1[i] += a1 * b1[i];`
			`  c2[i] += a2 * b1[i];`

partial unroll j, factor 4; partial unroll i, factor 4	`for (i = 0; i < NI ; i+=4)`
	`  for (j = 0; j < NJ ; j+=4)`
	`  for (k = 0; k < NK ; k++)`
	`  c[i][j] += a[i][k] * b[k][j];`
	`  c[i][j+1] += a[i][k] * b[k][j+1];`
	`  ...`
	`  c[i+3][j+3] += a[i+3][k] * b[k][j+3];`
		⇒	`for (i = 0; i < ni ; i++)`
			`  c1 += a1[i] * b1[i];`
			`  c2 += a1[i] * b2[i];`
			`  ...`
			`  c16 += a4[i] * b4[i];`

interchange i,k; partial unroll j.	`for (k = 0; k < NK ; k++)`
	`  for (j = 0; j < NJ ; j+=2)`
	`  for (i = 0; i < NI ; i++)`
	`  c[i][j] += a[i][k] * b[k][j];`
	`  c[i][j+1] += a[i][k] * b[k][j+1];`
		⇒	`for (i = 0; i < ni ; i++)`
			`  c1[i] += a1[i] * b1;`
			`  c2[i] += a1[i] * b2;`

Figure 3.4: Several transformed mini-mmm and their corresponding 1D kernel. The first is a kernel of 2 daxpys, the second is 16 dot products and the third is the same as the first (modulo commutativity of the multiplication and renaming).

```
for(i = 0 ; i < ni ; i++)
  c11 += a1[i] * b1[i];
  ...
  c1l += a1[i] * bl[i];
  c21 += a2[i] * b1[i];
  ...
  ckl += ak[i] * bl[i];
```

Figure 3.5: 1D Kernel: dot product k,l

```
for(i = 0 ; i < ni ; i++){
  c1[i] += a11 * b1[i];
  ...
  c1[i] += a1l * bl[i];
  c2[i] += a2l * b1[i];
  ...
  ck[i] += akl * bl[i];
```

Figure 3.6: 1D Kernel: daxpy k,l

```
for (i = 0; i < ni ; i++)
  for (j = 0; j < nj ; j++)
    c1[j] += a1[j] * b[i][j];
    ...
    ck[j] += ak[j] * b[i][j];
```

Figure 3.7: 2D Kernel: dgemv k

```
for (i = 0; i < ni ; i++)
  for (j = 0; j < nj ; j++)
    c[i][j] += a[i] * b1[j];
    ...
    c[i][j] += ak[i] * bk[j];
```

Figure 3.8: 2D Kernel: outer product k

3.4 Kernel Composition

The objective of the library generation is to choose the best performing code among the different possible decompositions, depending on input sizes and on individual kernel performance measures.

Kernel performance is measured independently of the application context. The advantages are that it is only necessary to run small kernels instead of the whole application, and performance measures can be reused many times. As a counterpart, we must use constant performance kernels and ensure that the kernel performance outside of the application context is the same as the kernel performance inside the application or function. This result is obtained in particular if the location of kernel inputs in the memory hierarchy is the same for the application as for the benchmark. Memory copies (or prefetches) are then required in order to reproduce the same performance in the application or library function.

At this point, multiple versions of a code are generated, and each version is decomposed into constant performance kernels and copies. As all inner loops are inside kernels, the few remaining statements that are neither kernels nor loops have an impact on performance that can be neglected. This implies that if we consider kernels and copies as the only statements that matter for performance, the architecture model is simplified:

- There is no ILP and memory instructions are blocking. Indeed, each kernel is in a separate function and are compiled separately. If we add copy operations for a kernel (block copies or bloc prefetch), then these operations are considered also as kernels and do not overlap with other copies or kernels for

the same reason. Copy statements can be considered as blocking memory instructions.

- All statements have a constant latency, depending on the location of its inputs in the memory hierarchy. In order to limit the number of possibilities for different configurations, we do not consider the case where input data is partly in cache, partly in memory for instance (or between two cache levels). These latencies are measured independently for each kernel and for each cache level. Dynamic mechanisms such as out-of-order mechanism reschedule instructions only in a limited window, much smaller than the number of instructions of kernels. Therefore they have a very small impact on the individual latency of a kernel. We assume that performance is additive between kernels, meaning that two consecutive kernels take as many cycles as the sum of the cycles taken by each kernel. In other words, provided the inputs of a statement stay in some cache level (or in memory), the statement latency is not contextual and does not depend on surrounding statements. Note that this simple assumption is wrong usually for simple assembly instructions.

This is a major simplification of the machine model. We now need a cache model in order to optimize latencies due to copies.

Moreover, to simplify further the performance model, we assume the cache is fully associative and has only one level. Therefore computing the best latency of a given code decomposed into kernels can be expressed with the following problem:

Pb. 4 (Composition of kernels) *Let M be an architecture with no ILP, blocking memory accesses, constant latencies, and an associative cache. For all program P that M can execute, is it possible to compute the minimum latency necessary to execute P on M?*

This problem is still an open issue. Answering this problem would enable to compute the minimum amount of copies necessary for the program. A similar problem would be to compute the minimal latency of a program, with spill/fill code (and constant latencies).

As finding the schedule with copies and the best latency is still a complex issue, we propose the following heuristic for static control programs. We assume the cache is fully associative (effective cache size can be measured with tools such as X-ray [166]). We first schedule immediately before each kernel a copy of its input towards the cache. Then, by hoisting, coalescing and removal of copies whenever there is some reuse, the total number of copies is decreased in order to improve performance. This simple technique relies on the computation of working set sizes. Working sets and iterations spaces (depending on input sizes) are used in order to determine whether these transformations on copies are possible. A Integer Linear System is built in order to compute the best schedule and type of copies, minimizing the latency according to problem size.

```
0  for (i=0; i<N; i++)
   |  for (j=0; j<128; j++)
1  |  |  X[i][j]=a*X[i][j]+Y[i][j]
   |  for (j=0; j<128; j++)
2  |  |  Y[i-1][j]=Y[i-1][j]+X[i][j]
```

(a) Initial code.

```
0  for (i=0; i<N; i++)
1  |  K1(X[i][0:127],Y[i][0:127])
2  |  K2(X[i][0:127],Y[i-1][0:127])
```

(b) Code representation with kernels and array sections.

Figure 3.9: Code example for copy equations. The purpose is to determine where to insert copies in (b).

Consider for instance the code of Figure 3.9. The problem is to schedule a copy or prefetch of the array sections accessed by the kernels in lines 1 and 2, with the following constraints:

- The copy or prefetch is scheduled before the execution of the kernel reading the array section,
- The data copied is still in cache when the kernel is executed. This constraints translates into the fact that the working set accessed between the copy and the read is smaller than the cache size (associative cache, no conflict misses).

We consider only a limited number of time steps to schedule these copies. Copies are simply inserted in the existing code just before kernels and they are hoisted and coalesced whenever the preceding constraints are true. 0-1 variables account for the textual position of the copies. In the following equations, x_2^2, x_2^{1d}, x_2^1 and x_2^0 are 0-1 variables representing possible position for the copy of `X` accessed in Statement 2. x_2^2 is equal to 1 if the copy is performed immediately before Statement 2. x_2^{1d} and x_2^1 are equal to 1 if the copy is instead just before Statement 1, the first in the case of a copy of a smaller section, due to the reuse of `X`. Finally, x_2^0 is equal to 1 if the copy is hoisted out of the loop. Likewise, x_1^n represent the different positions for the copies of `X` used in Statement 1, and y_p^n represent positions for the copies of `Y`. The following system is generated, where CS stands for the cache size, WS_P for the working set used in statement P and Big is a big number (a classical trick in ILP to make systems linear):

$$
\begin{array}{ll}
|WS_{1;2}| < CS + Big * (1 - x_2^1) & |WS_{1;2}| < CS + Big * (1 - y_2^1) \\
X[i][0..127] \subset WS_1 \Rightarrow x_2^{1d} = 1 & Y[i-1][0..127] \subset WS_1 \Rightarrow y_2^{1d} = 1 \\
|\bigcup_{i=0}^{N} WS_{1;2}(i)| < CS + Big * (1 - x_2^0) & |\bigcup_{i=0}^{N} WS_{1;2}(i)| < CS + Big * (1 - y_2^0) \\
|\bigcup_{i=0}^{N} WS_{1;2}(i)| < CS + Big * (1 - x_1^0) & |\bigcup_{i=0}^{N} WS_{1;2}(i)| < CS + Big * (1 - y_1^0) \\
x_1^0 + x_1^1 = z_1 & y_1^0 + y_1^1 = z_1 \\
x_2^0 + x_2^1 + x_2^{1d} + x_2^2 = z_1 & y_2^0 + y_2^1 + y_2^{1d} + y_2^2 = z_1 \\
z_1 = 1 &
\end{array}
$$

The program latency is expressed by the following function to minimize, where c correspond to latencies of copies or kernels:

$$
\begin{aligned}
cycle[0] \quad &= N * c_{kernel}(K1) + N * c_{kernel}(K2) \\
&+N * c_{copy}(X[i][0..127]) * x_1^1 + N * c_{copy}(Y[i][0..127]) * y_1^1 \\
&+N * c_{copy}(X[i][0..127]) * x_2^1 + N * c_{copy}(Y[i-1][0..127]) * y_2^1 \\
&+N * c_{copy}(X[i][0..127]) * x_2^2 + N * c_{copy}(Y[i-1][0..127]) * y_2^2) \\
&+c_{copy}(X[0..N][0..127]) * x_1^0 + c_{copy}(Y[0..N][0..127]) * y_1^0 \\
&+c_{copy}(X[0..N][0..127]) * x_2^0 + c_{copy}(Y[0..N][0..127]) * y_2^0
\end{aligned}
$$

The constraints of the system are all affine but the objective function usually is not. Indeed, while in all constraints, the coefficients of variables x_m^l and y_m^l are always constant (Big or 1) and the working sets are parametric expression, in the objective function, the volume of the iteration domains are coefficient of the variables. Working sets and iteration domain sizes are positive parametric polynomials, that can denoted exactly with Ehrhart polynomials (as described by Clauss [32]). We propose to build a decision tree sorting out the coefficients of the objective function. By ordering the coefficient by decreasing order into a vector, we build a new linear vector cost function, to minimize lexicographically. A parametric solver (PIP[58], Omega[84]) computes then the optimal solution according to the input sizes. The resulting code of the example, with the copies is shown in Figure 3.10.

```
if (256*N+128>2048) then
    cpy(Y[-1][0..127])
    for (i=0; i<N; i++)
        cpy(X[i][0:127])
        cpy(Y[i][0:127])
        K1(X[i][0:127], Y[i][0:127])
        K2(X[i][0:127], Y[i-1][0:127])
else

    for (i=0; i<N; i++)
        K1(X[i][0:127], Y[i][0:127])
        K2(X[i][0:127], Y[i-1][0:127])
```

Figure 3.10: Optimized code with two versions for copy scheduling depending on N.

Tiling introduces a new constraint in the previous system, stating that the working set of the tile fits into the cache. Usually, the exact size of the tile (especially when the tile has many dimensions) is not known.

Making the tile size one of the variable of the problem makes the previous system non affine. Indeed, all working sets can be polyhedra defined by input parameters and tile sizes. The volume of these working sets are polynomials and the system of Diophantine equations it creates cannot be solved parametrically. We resort to a simple enumeration of possible tile sizes (replacing each tile size by a possible constant value) so that the working set of the tile fits into the cache and the system can be solved. Multiple cache levels add one set of constraints (constraints for each cache size) for each level. The constraints are generated in the same manner. Multiple tiling levels however increases the complexity of the search for an optimal solution.

3.5 Application on Linear Algebra Codes

Different codes from dense linear algebra have been targeted for validating our approach:

- 1 loop: vector dot product, DAXPY, 1D convolution
- 2 nested loops: matrix vector multiply (DGEMV)
- 3 nested loops: matrix matrix multiply (DGEMM) and Cholesky resolution (POTRES).

All of these codes are part of BLAS library, with the exception of POTRES. Notice that the selected functions of BLAS are representative of all BLAS functions, since DGEMM is used for instance in many BLAS3 functions. Multiple versions of linear algebra functions are generated using X-language. A special pragma is used to trigger the decomposition into kernels:

```
#pragma xlang decompose n
```

This directive decomposes the considered code fragment into kernels of depth n. For instance, when $n = 1$, kernels only correspond to one single loop. The automatic selection of the best kernel according to the tile size is not yet automatic.

On the hardware side, two different architectures have been used:

- Itanium 2: BULL Novascale server featuring 1.6GHz Itanium 2 processor, with 3 cache levels was used. Out of these 3 levels, only the 2nd level (256KB, unified) and the 3rd level (9MB, unified) can contain floating point values. The processor offers 128 floating point registers and can issue up to 6 instructions per cycle.
- Pentium 4: PC equipped with an Intel Pentium 4 Prescott 2.80GHz processor with two cache levels: L1 D cache (16 KB) and L2 (1MB, unified) was used. The processor can issue up to 2 instructions per cycle and supports the SSE2 extensions. The SSE2 instructions allows to vectorize most of the standard floating operations (up to 2 double precision word can be packed in a single instruction). On this Pentium version, the number of registers available for SSE2 is limited to 8 making register allocation a sensitive issue.

Our kernels and codes were compiled using Intel ICC Compiler v9.0 on both platforms. The code generated with our approach was compared to the MKL library (version 9.0), IPP library (version 5.1) and ATLAS library generator (version 3.6), the same release number being used for both machines.

Kernel Performance Analysis

The hierarchical compilation method does not require to analyze the performance of each kernel. We could just benchmark them and schedule the copies in order to generate a library. However, the approach relies on the compiler to generate high performance kernels. If the compiler fails to do so, it could be interesting the understand why and how to palliate to this failure. Moreover, decomposition of different codes often lead to the same kernels. As they represent the code fragments where functions spend most of their execution time, it is interesting also to learn the behavior of the machine and of the compiler on these kernels. These ideas are used and developed in the following chapter.

We will focus on the some key kernels representative of kernel performance analysis. Since all of our kernels (see Figures 3.5, 3.6, 3.7 and 3.8) correspond in fact to rectangular matrix multiply operation, we also compared our kernels to MKL and ATLAS.

A dotproduct-k, k kernel of length n has a fairly small working set: $2 \times k \times n + k \times k$. If there is enough register space, it offers also a very good memory/arithmetic ratio: $2 \times k$ loads for $k \times k$ multiply add. Increasing k will decrease the stress on load instruction scheduling but at the same time will increase register pressure up to the point where the register allocator will have to insert expensive spill/fill instructions or to reload some of the operands. Results for the Itanium for $k = 4$ are presented in Figure 3.11. For all vector lengths the working set is small enough so all of the data is contained the L2 cache. Increasing k improves performance because kernels become more and more floating point dominated. Dotproduct-$4, 4$ offers peak performance around 6.2 Gflops but this requires vector length of at least 200 (however using larger vector length does not generate performance loss). However, for $k = 8$ (not shown here), the register pressure is too high: the compiler start inserting spill/fill instructions and furthermore, the loop body becoming too complex, the compiler can no longer software pipeline the loop and the performance drops to less than 5 Gflops.

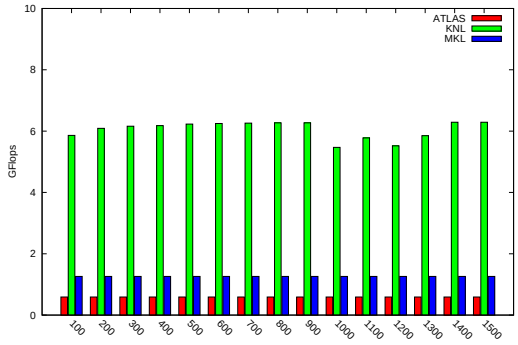


Figure 3.11: Dot product kernel performance on Itanium2

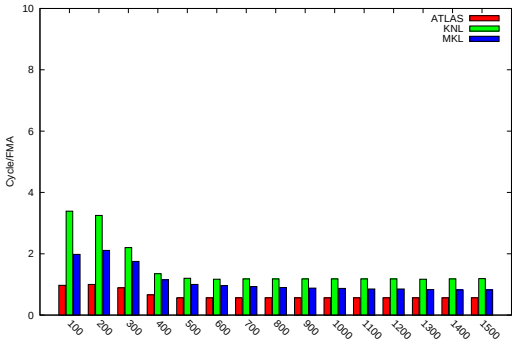


Figure 3.12: Outer product kernel performance on Pentium4

An outerproduct-k kernel of length n has a larger working set: $2 \times k \times n + n \times n$. The memory/arithmetic ratio is good: $2 \times n \times k$ loads for $n \times n$ stores and $n \times n$ multiply adds, but requires large temporary storage (much larger than register space). However, it should be noted that register pressure will increase quickly in function of k but also in function of n, therefore data will have to reloaded (but fortunately most of the time from cache). On Pentium 4 (see Figure 3.12), the performance drops for $n = 300$, the working set exceeds L2 cache size. Now increasing k, improves operand reuse in L1 but since L1 is fairly small, already $k = 8$ exceeds L1 capacity level. However, it should be noted that outerproduct-4 kernel on Pentium performs much better than all of the dotproduct-k, k kernels.

Library Generation

We select in this step the best performing version, with its necessary copies or transpositions, according to the size of input data. The final result is generally a function that selects with a decision tree the best version. We show here only two results for DGEMM function from BLAS3 and POTRES from LAPACK. Other results on 1D convolution or DGEMV can be found in [16].

The matrix multiply function can be decomposed in several ways, presented in section 3.3. The array in Figure 3.13 shows the performance of codes obtained by composition of the best kernels and their copies, according to the input size N for square matrices. In order to compare the relative performance, all cycle counts are divided by N^3, the complexity of the algorithm. These cycle counts are predicted according to the previous model, and a decision tree is built in order to select the best kernel decomposition for any given input value N.

N	Kernel name	Kernel perf. estimation (cycle/N^3)	Copy perf. estimation (cycle/N^3)	Total perf. estimation (cycle/N^3)
100	nkkn_100_4_100	0.53	0.02	0.55
200	knnk_4_200_4	0.52	0.01	0.53
300	knnk_6_300_6	0.51	0.02	0.53
400	knnk_4_200_4	0.52	0.01	0.53
500	knnk_4_500_4	0.51	0.014	0.52
600	knnk_6_300_6	0.51	0.015	0.53
700	knnk_4_700_4	0.51	0.02	0.53
800	knnk_4_800_4	0.51	0.027	0.54

Figure 3.13: Decomposition of square matrix multiply (NxN)x(NxN) into different kernels according to input size N, for Itanium2. For each value of N, the name of the best performing kernel, the predicted time in cycle/N^3 spent in the kernels, spent in the copy and total cycle count are provided.

For a matrix product, 95 different kernels were benchmarked. 11 of them are used for the matrix multiply version between square matrices. The overall performance of the function is shown in Figure 3.14, with L2/L3 cache miss counts. Notice that due to the copies and the tiling, the performance does not degrade when the number of cache misses increase. Moreover, the measured performance stays within 1% of the performance predicted by our model (the two curves are one over the other).

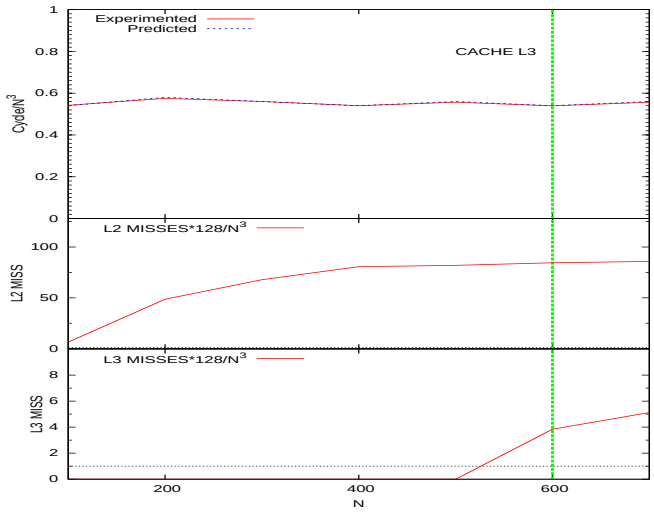


Figure 3.14: Performance of DGEMM in cycle/N^3 superposed with the number of L2 and L3 cache misses/N^3. The code is made of 11 kernels, chosen according to N.

The resulting performance on the whole square matrix multiply is presented in Figure 3.15. In all cases, our approach (KNL) outperforms ATLAS and reduces the gap with the Intel MKL library. It is interesting to note that on the Pentium 4, performance obtained still lags behind MKL. The main explanation is that we did not succeed in making the compiler make the best use of 8 SSE2 registers. Different kernels are used for different sizes. For the rectangular matrix multiply, our results are much better than Intel MKL library because the kernels selected are adapted for small sizes.

For *POTRS* function, we used fission, interchange, strip-mine and unroll in order to generate constant performance kernels. The main difficulty of this code comes from loop carried dependences. The following code is an extract of *POTRS* code. Performance results are presented in Figure 3.16 and our approach

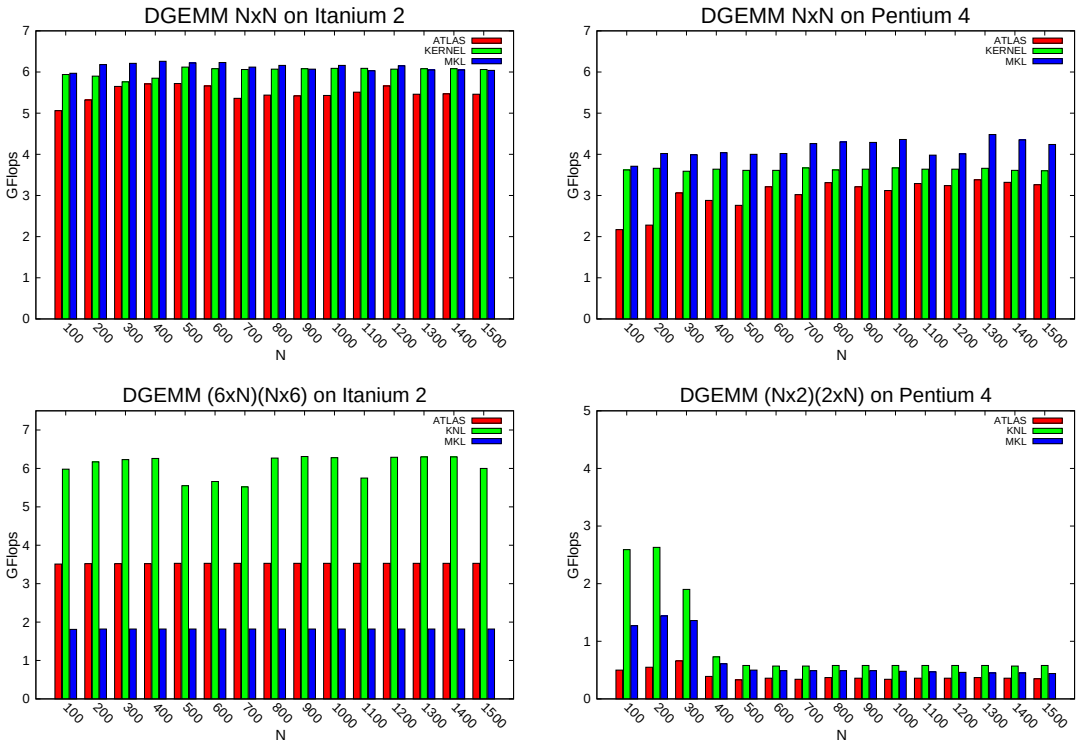


Figure 3.15: DGEMM kernel performance on Itanium 2 for square (NxN)x(NxN) and thin matrices (6xN)x(Nx6) (left) and on Pentium 4 for square (NxN)x(NxN) and tall matrices (Nx2)x(2xN) (right), for ATLAS, MKL and our code (KNL).

exhibits around 20% speed-up on the naive code, and matches (on Pentium4) or outperforms (on Itanium2) the Intel IPP library function. Again, it shows that our approach is well-suited for all input data sizes, since the decomposition and the kernels used are chosen according to the input size.

3.6 Application on Large Applications

We show here how hierarchical compilation can be applied to large applications in order to optimize their performance. Instead of applying the method on library functions, we apply it on the hot functions of the application (where most of the execution time is spent). The decomposition then generates kernel functions, these functions defining a high-performance library for the application. While this may be surprising to call this set of kernels a library, the kernels indeed correspond to codes that are independent of the application itself and we show on our examples that they are sometimes reused multiple times either in the same application, or in other applications/libraries. The major difference with library functions is that the decision tree selecting the best decomposition, according to input array sizes, is pruned according to profiling information. This is still on-going work, the only publications being the deliverable of the project.

The applications considered are those considered in the ANR project PARA [118]:

- BLAST: This is a well-known code in genomic, comparing genomes to genes. The version considered for decomposition is not the standard/official version, developed by the NCBI, but the one developed by IRISA/Symbiose (it delivers the same results but the algorithm used is different). The details of the optimization are presented in deliverable T7.1 [14].

```
for (j=N; j>=1; --j)
  for (k=M; k>=1; --k) {
     B[k][j] = alpha*B[k][j]
     B[k][j] /= A[k][k]
     for (i=0; i<k-1; i++)
       B[i][j] -=  A[i][k]*B[k][j]
  }
```

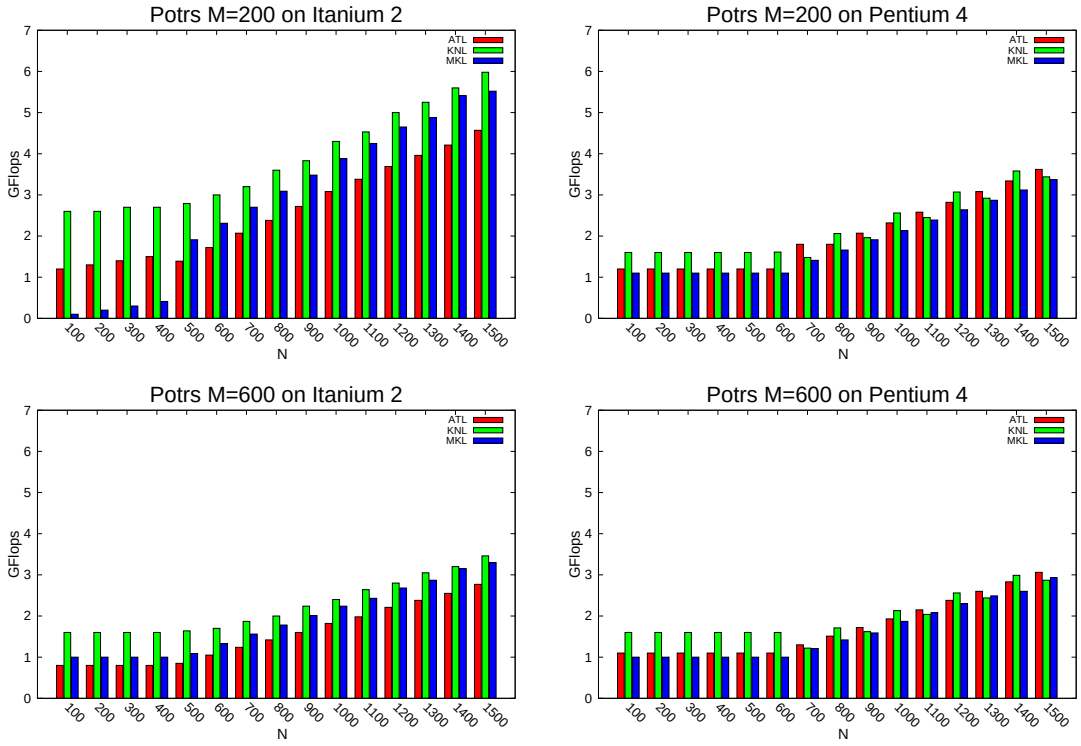


Figure 3.16: LAPACK POTRS on Itanium2 (left) and on Pentium4 (right) with M=200 and M=600, for ATLAS, MKL and our code (KNL).

- LQCD application: this application is representative of Lattice Quantic ChromoDynamic applications. The code considered, HMC (for Hybrid Monte-Carlo), is the result of the ETMC collaboration. The optimization effort is in collaboration with the laboratory of theoretical physics (LPT) of Orsay, IRISA, INRIA-Futurs of Orsay and the University of Versailles. The details of the preliminary optimization are presented in deliverable T5.4 [13].

- GIBBS: this application, developed by the Institut Français du Petrole (IFP) and the laboratory of chemistry of Orsay is a molecular simulation. Its objective is to compute thermodynamic properties of fluids using a precise simulation of a few hundred molecules, based on a Monte Carlo method. The details of the optimization are presented in deliverable T5.3 [81].

These applications exhibit some common features: they are not static control programs, they use arrays indexed by arrays and whiles (or ifs with non-static predicates), and one hot function takes most of the execution time.

We performed a hierarchical kernel decomposition of these hot functions. To do so, we had to consider in addition to fusion, interchange and unroll, other transformations: fission (or loop distribution), deep jam [29], vectorization and slicing. Deep jam corresponds to an unroll and jam applied an outer loop, with a while as the inner loop. Slicing is used to separate, through fission, statements of a loop contributing to the

computation of different expressions. The advantage of slicing is that is simplifies the loop size and does not create additional array temporaries (as this may happen for fission). Up to now, we have not tried to generate automatically kernels from X-language pragmas, as some extensions of X-language transformations are required. The hot functions of the applications have been decomposed into two types of kernels:

- constant performance kernels,
- codes with for loops and assignments, constant loop bounds and strides, constant array sizes, and array indices are array values (like `a[b[i]]`).

For the second type of kernel, its performance only depends on the configuration of its input data in the memory hierarchy, as for constant performance kernels. This is however more difficult to control.

For instance, for the LQCD code, the hot function, called `Hopping_Matrix`, has more than 200 loc and can be decomposed into 3 kernels for instance. It means that most of the execution time of the function is spent in these kernels. This is only one possible decomposition, but in all cases and this can be observed for

```
void k1(su3 * iU,
        spinor * s,
        halfspinor * restrict *phi,
        complex ka)
  int i;
  su3_vector psi,chi;
  for(i = 0; i < N; i++)
    _vector_add(psi, s[i].s0, s[i].s2);
    _su3_multiply(chi,iU[i],psi);
    _complex_times_vector(phi[8*i]->s0, ka, chi);
```

(a)

```
void k2(spinor *s,
        halfspinor **phi)
  int i;
  for(i = 0; i < N; i++)
    _vector_sub(phi[8*i]->s0, s[i].s0, s[i].s2);
```

(b)

```
void k3(su3 * iU,
        spinor * s,
        halfspinor **phi,
        complex ka)
  int i;
  su3_vector psi,chi,mu;
  for(i = 0; i < N; i++)
    _su3_inverse_multiply(chi,iU[i],phi[8*i]->s0);
    _complexcjg_times_vector(psi,ka,chi);
    _vector_add_assign(s[i].s0,phi[8*i]->s0);
    _vector_add_assign(s[i].s0,psi);
```

(c)

Figure 3.17: Some kernels obtained for LQCD code, Hopping_matrix function. The entire function can be decomposed into these three kernels. Statements correspond to macros on 4 dimensional complex vectors and `N` is a constant.

the other applications:

- Kernel generation is achieved by hand but their automatic generation does not seem out of reach. We will discuss possible extensions to X-language in the conclusion.
- The kernels generated usually do not correspond to known library functions, even if in some cases (GIBBS), BLAS-1 functions can be recognized.
- The kernels exhibit good performance compared to the original code (and the compiler generates a good quality code). The overall speed-up will be discussed in the following section.
- The same kernels appear in different places of the code, as in the LQCD application. This form of reuse decreases the cost of kernel benchmarking.

For performance, we report here the current results obtained for these applications. Some optimizations are still on-going work and the target machine is an 1.6Ghz Itanium-2 with 256KB L2 and 9MB L3 caches.

- GIBBS: a speed-up of 2.24 was obtained for a representative simulation with one molecule, by optimization of `energy_tot_LJ` function code. The larger part of the speed-up comes from specialization on iteration count that implies a more efficient and simpler kernel decomposition.

- LQCD: the first results show a speed-up of 30% for a representative lattice of size $16^3 * 32$, by optimization of `Hopping_Matrix` function code.

- BLAST: the application developed at IRISA is in two passes. Optimization focuses only on the first one (the most time consuming). For various input data sets, the hierarchical compilation using deep jam and vectorization speeds up the application by a factor of 2.

For all these applications, we checked the assembly codes generated for optimization opportunities with our tool, MAQAO (presented in following chapter).

While the hierarchical compilation is not yet automatic for general applications, it shows that designing library functions at the same time than application optimization is a promising path to obtain high and stable performance codes. This advocates also for the use of automatic recognition of already tuned kernels, with methods such as those presented in the previous chapter, that may not correspond to predefined algorithmic computations.

3.7 Related Works

Among related works, many works have been dedicated to iteration exploration of optimization search:

ATLAS [157] explores tile sizes and performs some simple micro-optimization (software pipeline, scalar promotion,...), but it mainly relies on only one kernel. This kernel was chosen according to its good ratio memory accesses/computations, not according to its performance on the target architecture. It is however possible to introduce new high performance kernels into Atlas, since there is an add-on mechanism that enables Atlas to use external, possibly hand-tuned assembly codes. Compared with Atlas, the approach described in the paper is not limited to specific application and performs quite extensive search for the micro-optimizations, having the opportunity to find better kernels. This appears on the performance results, where our approach compares to vendor library performance and outperforms Atlas. On the other hand, our method relies on the programmer to find out the size of the cache or the maximum size of the tiles.

For model-based ATLAS [164], the model targets essentially cache behavior. Our approach focuses more on micro-mmm optimization, and resorts to simple model based tiling and then iterative search for finding tile sizes, guided by the user. The use of more complex models (see for instance Cascaval and Padua for stack distances [25] or Fraguela *et al.*[61] for a more complex model) is still possible.

Extensive search among optimizations[42] shows that it is difficult to understand the links between optimization parameters, optimization sequence and performance. The exploration proposed by the authors is very time consuming and yet does not include many optimizations. In comparison, our method resorts to a very small number of transformations and relies on existing compiler to perform adapted optimizations.

The compiler optimization space exploration proposed by [150] changes the heuristic guiding optimizations by a search. This search is not exhaustive and is guided by a cost function. The goal is mostly to improve the optimization step of the compiler but does not seem to be aggressive enough to apply to library optimization.

Finally, [70] describes a methodology for hand-tuned optimization, applied to BLAS optimization. The authors propose a decomposition of micro kernel similar to ours, according to different tile sizes. The main focus of this work is to compact the data layout (making copies or transpositions of arrays) in order to improve TLB hit ratio. All the fine-tuning of micro kernels is however performed by hand. In comparison, our approach is automatic, at the expense of a small performance degradation, and is not specific to matrix matrix multiplication.

3.8 Conclusion

This chapter has proposed two approaches to tackle the difficult problem of automatic high-performance library generation. The two approaches rely on empirical search.

For the first one, the user defines explicitly the search space of optimization sequences and parameters. This space is described by a language of pragma directives, X-language, decorating the source code. This approach is very flexible, runs with any compiler, and unlike other approaches such as ATLAS or FFTW, the meta-compilation framework based on X-language is not application-dependent. However, applying X-language directives on naive BLAS codes in order to generate a high performance library has shown also a number of limitations:

- Long sequences of transformations are difficult to handle and are error-prone. This is mainly due to the fact that in our implementation, there is no validity check applied on the transformations. This is a main issue in all languages and this could be addressed here by validating either each individual transformation or only the whole sequence transformation. For the first one, correctness tests are conservative and may disable some transformations that are valid in the application context. The second one does not suffer from this limitation (sequences of invalid transformations can be valid), however debugging is more difficult. The work of [151] has presented interesting results on this method.

- The code selected by experimental search corresponds to one of the versions generated during the search. It implies that the version executed does not depend on input data (in particular, data size). This is clearly a suboptimal approach since libraries should be able to deal efficiently with any of its input values. Ideally, we would like to choose the best version for a given input data. For static control programs, this implies a decision tree based on input vector sizes. Defining a decision tree with pragmas is unpractical and may lead to combinatorial explosion (each leaf has its own search space). Moreover, defining the value from which this version is selected instead of that one is not obvious and may require exhaustive search.

Note that this second shortcoming is shared by most other experimental search-based techniques.

The second approach is based on a hierarchical decomposition of the codes: the codes are transformed so that their statements are constant performance kernels. While the kernels focus on optimizing ILP, the code using the kernels optimizes locality. Generating different decomposition therefore finds the best tradeoff between ILP and locality. Moreover, experimental search is applied only on the constant performance kernels. This entails several advantages:

- There is no need to run the whole function. Only small fragments are benchmarked, implying faster tuning time. Moreover, the same constant performance kernels are reused in different function decompositions, leading to further gain in benchmarking time.

- Performance evaluation of a decomposed function can be done by a model and can be accurate. Indeed, after decomposition, function statements are either constant performance kernels or explicit copies of input data to cache. Kernels and copies are executed sequentially, and the machine can be simplified into a sequential machine with constant latency instructions and a cache hierarchy (no vector instruction, no ILP, no super scalar mechanism since they are all taken care of at the kernel level). While this model is still complex, we have shown that some simple performance model can predict accurately linear algebra code performance.

- From the multiple decompositions, an optimized function is generated, selecting the best version according to input sizes. Thus a multi-versioned function, exhibiting performance for a wide range of input data, is generated from a few performance measures of kernels and copies. Using both performance model and experimental search on constant performance kernels is one of the main assets of this approach.

The hierarchical decomposition approach has been successfully validated for linear algebra routines, on Itanium and Pentium architectures, outperforming ATLAS and being very competitive with vendor-routines

(MKL and IPP from Intel). In particular, one distinctive feature of our approach is its ability to generate high performance code for all input sizes, outperforming vendor-routines for all extreme cases (thin matrices in a matrix multiply, for instance). One of the strength of the hierarchical decomposition, resorting only to source-to-source transformations and to a production compiler is also its weakness: we rely on a "good" compiler to achieve high performance on our building blocks/kernels. As a matter of test, we used an approach replacing ICC with GCC. On the Itanium 2, the performance results were much lower while on the Pentium the performance gap was smaller. This clearly shows the necessity of having a very good compiler for the kernels. Since the kernels used are fairly simple (static control programs with constant loop bounds and constant array sizes), a specialized "compiler" for such kernels can be developed. A good example of such a compiler is the XLG [28] tool developed by CAPS Enterprise. Another approach, developed in the following chapter, is to evaluate the quality of the assembly code generated by the compiler in order to decide whether there are still optimization opportunities or not. Assessing statically the performance of assembly code is also another way to reduce benchmarking time.

Moreover, our approach as been successfully applied to more general applications (chromodynamic application, BLAST, molecular simulation), exhibiting significant speed-up. These applications are irregular codes. The decomposition of the hot functions required transformations such as deep jam and fission. While the method is not fully automated yet for this kind of code, we have shown that these functions could be decomposed into kernels depending only on the position of their input in memory hierarchy (similar to our definition of constant performance kernels). The optimized function is built from a selection of the best decomposition codes, according to the value (obtained through profiling) of its inputs. We obtained significant speed-ups on Itanium architecture.

Hierarchical decomposition is a promising approach for optimization of both libraries and applications. For applications, the method generates a library of all the kernels used in performance-critical functions and optimizes the application by using this library. Having several applications in a domain, the method would then design a library of kernels that are (re)used in this domain. This library co-design mechanism – designing the library and optimizing library and application at the same time – could easily be combined with the approach of the previous chapter. A natural method would be to resort first to template recognition of known library functions, trying to match as many code fragments as possible to library functions. When some code fragments do not match or cannot be substituted by library calls, we then resort to the hierarchical decomposition method, that creates new library functions. The main advantage of the combination of the two methods is that the larger the library grows (by adding new kernels), the smaller the experimental search of the hierarchical decomposition. This will be the subject of future work.

The main extension of this work is its extension to parallel codes, and in particular to multithread codes (shared memory model). Assuming some task parallelism has been detected, one of the main issue of high-performance parallel code generation comes from scalability: usually, the best performing parallel code does not correspond to the parallel execution of the best sequential tasks. As all tasks share some path to memory, limitations on the memory bandwidth may degrade performance of individual tasks (even with embarrassingly parallelism). Cache coherence, out of order loads and stores, synchronizations are some of the mechanisms that make performance prediction of a parallel code difficult, even when the performance prediction of the sequential codes is accurate. Automatic generation of constant performance parallel kernels that would be the building blocks of high performance parallel applications is still an open question.

Chapter 4

Assessing and Improving Compiler Generated Code Quality

This chapter focuses on methods and tools that assess compiler-generated code quality and performance. We first present analysis techniques and tool for performance evaluation and then describe two methods for assembly to assembly optimizations.

4.1 Introduction

The quality of a code produced by a compiler is essential to get high performance. In the old CISC days, quality could be simply assessed by counting the number of instructions. Nowadays, with the recent generation of microprocessors, such simple metrics are no longer valid. First of all, caches have introduced data locality as a key performance metric. But code quality also comes from the appropriate use of instructions such as branches, fused multiply add `-fma-`, predicated instructions or prefetches managing the memory hierarchy. Finally, some low level architecture mechanisms such as bank conflicts or load/store queue wrong aliasing have an impact on execution time and require to analyze in detail memory address streams.

In fact, on modern microprocessors, taking into account all of the architectural features is critical to get high performance. For example on Itanium systems, *gcc* which performs most of the standard (architectural neutral) optimizations is very often outperformed by Intel C Compiler (*icc*) because *icc* considers all of the features offered by the Itanium 2. However this performance comes at the expense of code complexity and stability: the code generated is difficult to analyze and optimizations are often unstable, meaning that a slight modification on the source code, or on some input value, may change drastically the overall performance. Due to this complexity in the optimization process (phase ordering problem, heuristics used instead of exact methods, $k-$limiting techniques), some codes can be compiled into codes with important deficiencies w.r.t. performance (missing prefetches, non-pipelined loops, poor register allocation,...).

The consequence is that analyzing performance, deciding whether the optimization are appropriate or not for the input sets considered, finding possible performance bottlenecks and re-write the code accordingly (either at source or assembly level) is a necessity in the development process of high performance codes. We propose in this chapter a combination of static and dynamic assembly code performance analyses, building up an expert system guiding the user in the performance tuning process or providing feed-back information for an iterative compilation process. This expert system has been implemented in a tool, named MAQAO (*Modular Assembly Quality Analyzer and Optimizer*).

As shown in the introduction chapter, the difficulty to generate assembly code of good quality is not only a engineering issue, that may be solved by spending more time in the development of compilers. The root of the problem is a theoretical problem, and part of the difficulty comes from the fact that optimizations sequences and parameters must take into account parameters such as loop trip count for generating efficient code. For example, short loop trip count would favor full unrolling while very large loop trip counts will favor

deep software pipelining. Therefore the code generated has to be specialized depending upon these parameter ranges and then has to use extensive versioning to apply these different specialized versions. The classical drawback of such an optimization scheme is code expansion and decision tree overhead. It usually puts a hard limit on the total number of different specialized versions generated. To obtain better code quality, a modern compiler often needs to take into account input data sets, and then to resort to the analysis of previous runs (see [26] for value-profiling optimizations for instance).

We propose two new approaches for code specialization, both approaches tackling the problem of versioning overhead and code size for codes handling large input data sets:

- Composition of specialized versions [45]: For loop structures, we propose a new method to specialize code at the assembly level and to drastically cut the overhead cost with a new folding approach. Taking the assembly code, we are able for instance to generate three versions tuned for small, medium and large iteration number. We combine all these versions into a code that switches smoothly from one to the other while the iteration count increases. Hence, the resulting code achieves the same level of performance as each version on its specific iteration interval.

- Hybrid specialization [87, 88]: Code specialization, which works by substituting a formal input value by an effective value, can be done either statically or dynamically. Static specialization makes use of data that is expected to be frequently used and the compilation time does not impact code performance. On the contrary, dynamic specialization uses the actual values at run-time but the optimizations taking advantage of these values burden the execution time. We propose a novel hybrid method combining the best of both static and dynamic specializations, achieving most of the compilation work at static compile time with a lightweight dynamic specializer.

Section 4.2 describes the techniques used in our tool MAQAO in order to analyze, compare performance and guide optimizations. Section 4.3 proposes a motivating example for the two specializations approaches that are described in Sections 4.4 and 4.5. Finally, Section 4.6 presents the related works on performance analysis tools and on specialization techniques.

4.2 Modular Assembly Quality Analyzer and Optimizer

MAQAO analyzes assembly codes (with debugging information in order to make the link to the source code) and is able to instrument the assembly in order to perform dynamic analyses. Our goals can be summarized by the following:

- Automated extraction of information from assembly code;
- Combined advantages of static and dynamic analysis of assembly code;
- Intelligent navigation and flexible analysis through query on assembly code;
- Performance aware assembly development platform;
- Feed-back tool for user to double-check if the compiler did a good job;
- Evolving database of known performance issues on the target architecture (currently Itanium 2).

Some contributions of MAQAO need to be underscored as well:

- Database of known performance problems and automatic detection of these problems on any new code. Performance problems can be identified either by micro-benchmarking techniques on the target architecture[83] or by the user himself. This is addressing a problem where currently most of the optimization knowledge is informally stored (if not only in the programmer's brain). This productivity issue is not handled by other tools.
- Storing both high level code structure (loops, CFG) and low level profiling information (hardware counters, value profiling) in a database to perform requests on it is an important and novel aspect of code analysis. MAQAO therefore proposes the first building blocks of an assembly optimizer/analyzer. Case studies advocate for this approach which is not only less time consuming than a pure dynamic analysis method, but which is also able to detect in a more intuitive way the causes of inefficiency.

- Value profiling is addressing an important aspect (lacking on other tools), the link between observed behavior on the processor and software behavior. Section 7.1.2 shows that this can lead to large performance gains.

MAQAO is based on three distinctive steps: static analysis and restructuration of the assembly code, dynamic analysis of the program behavior, and reporting.

Static Assembly Analyses/Restructuring

MAQAO analyses assembly code in order to evaluate performance and code quality. The objective is to detect performance issues and to compute an optimistic performance bound to compare to dynamic results. MAQAO first restructures an assembly code and builds the structures:

- Call Graph: this structure slices the code into different functions and shows how they interact each other. For a given function, the number of calls to another function is annotated on the edge going from the caller to the callee.
- Control Flow Graph: the control flow graph structures basic blocks within a function, showing possible control paths. This information is important for loop detection and provides a quick assessment of the function complexity.
- Dependence graph: dependences are computed between statements, based on register reuse. This analysis is useful to validate future code transformation and optimizations. Moreover, it helps in computing, for some cases, prefetch distances.
- Loops: the loops determine basic blocks that are likely to be the most executed. Hence, loops are a legitimate focus for analysis and optimization.
- Bundles and Basic Blocks: Bundles are vectors of instruction on an EPIC VLIW.

This information can be accessed in batch mode or in a interactive mode, in which case the user can navigate directly from the graphs displayed and back and forth from the source and assembly codes. Figure 4.1 represents the control flow graph of a daxpy routine shown in MAQAO. The selected basic block corresponds to the highlighted assembly code on the main window. Notice that for a simple daxpy function, the compiler has generated a complex control flow graph with several versions depending on the alignment of data on 16B boundaries. The resulting code has more than 1K lines of code, advocating for a structured and automated approach such as MAQAO.

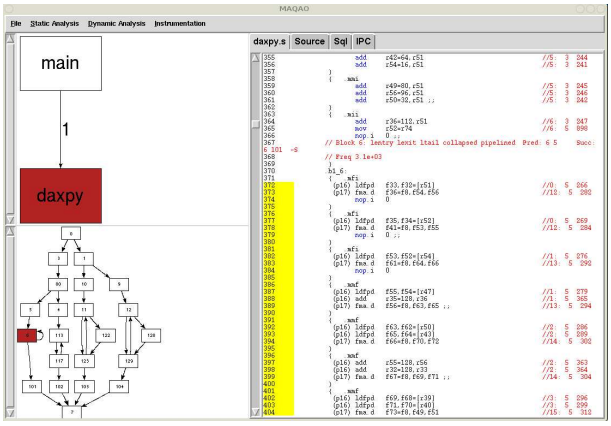


Figure 4.1: CFG and assembly code of a daxpy routine in MAQAO

A preliminary static performance analysis is achieved on the assembly code, starting on blocks and then inner loops. The performance model relies on two estimations: a cycle count (time taken by the loop) and an issue count (number of scheduled bundles). The cycle count is based on fixed latencies and takes into

account the parallelism inside bundles and the dependences between bundles, due to register reuse. Actually, this low level performance model for basic blocks is provided by the compiler (`ICC` in this case). Latencies for load operations are assumed to take a mean value between L2 and L3 latencies (the compiler schedules the code according to this latency). This means that cache miss effects are not handled. A more parametric performance model, taking into account other load latencies, is under study. The latency of each inner loop is thus estimated, taking into account the real schedule generated by the compiler, including possible bubbles. For the issue count, it helps in defining a simple lower bound latency for the loop, based on the idea of Boyd *et al.*[22]: the lower bound is computed by counting the minimal number of cycles required for the execution of the instructions in the loop body on the different functional units, assuming there is no dependence between instructions. Compared to software pipelining notations, the issue bound is known as the recII bound whereas the cycle bound is known as resII bound. These performance estimations, as well as the count of the different classes of instructions, are the building blocks of the hierarchical reporting system.

Instrumentation and Dynamic Analyses

Dynamic analyses help in focusing on specific code behavior offering optimization opportunities, such as specialization for instance. MAQAO offers the possibility to drive assembly code instrumentation at different level of the code structure. This low level technique has several advantages:

- It does not alter the behavior of the compiler, neither for the code generation nor for the code optimization, since the instrumentation is done as a post-compilation transformation (this is not the case for instance for `gprof`). It is therefore possible to instrument code versions that cannot be accessed at the source level. For instance, the compiler can generate many different versions for one loop. Computing the loop trip count distribution of the different generated loop versions is only possible at the assembly level. Likewise, profiling the behavior of an inlined function in a specific call site is only possible at the assembly level.

- The run-time overhead is kept small. This is done by injecting a limited number of extra-bundles, named *assembly probes*, around the targeted code fragments to monitor. These bundles are in charge of storing some specific registers in a dedicated memory zone.

The instrumentation can be done at function level, loop level, basic block level or instruction level. For all these levels of instrumentation MAQAO monitors and stores the value of the clock register and builds execution time profile. Additionally MAQAO is able to store every register value manipulated by the code and dynamically process it. Monitoring values manipulated by a binary at run time is often referred to as *value profiling* [26]. Value profiling is one of the key features of MAQAO, and it is often the missing link between the observed behavior on the hardware and the nature of the application. This feature yields to numerous optimization opportunities. Several scenario for code instrumentation are proposed:

Hot paths Identifying the path at run-time which crosses the whole program where the application spends the most of its time is a key element for understanding application behavior [94, 123]. An example of hotpath information displayed by MAQAO can be found in Figure 4.2.

Loop level profiling Both the time taken by the execution of the loop and the iteration count are captured by the instrumentation. A histogram of the iteration count values is computed dynamically. Figure 4.3 details results gathered by instrumentation at loop level of the SPEC FP 2000 [76]. These graphs depict the execution time distribution per loop depending on the total number of iterations. Histograms summarize iteration weight for an interval in power of two: $]0,1]$,$]1,2]$,$]2,4]$,$]4,8]$ and so on. For instance the bar labeled 64 coalesces all loops with a total number of iterations between]33,64]. Each benchmark is considered individually without weight related to its execution time. 4.3(b) presents the same data but in a cumulative histogram which is convenient to grab the slope of loop distribution. From these two graphs it can be said than Spec benchmarks spend 25% of their loop time within loops of less than 8 iterations. These numbers provide insight about code specialization opportunities for short loops where software pipeline and other unrolling techniques are often low-performers. In fact, this

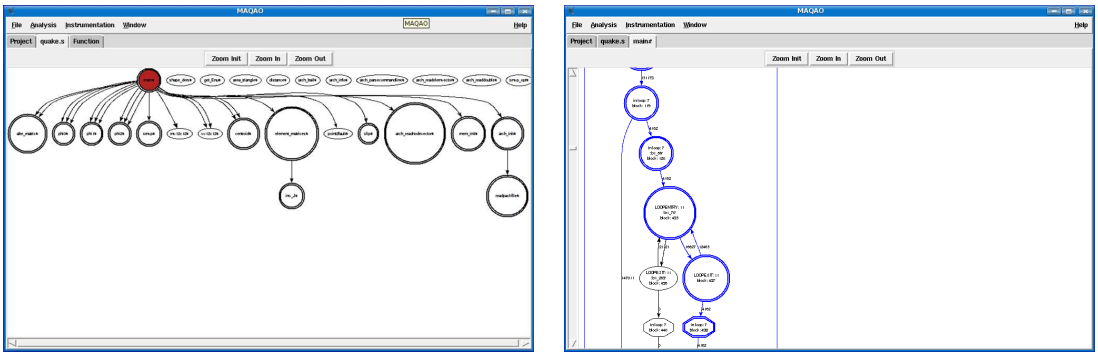
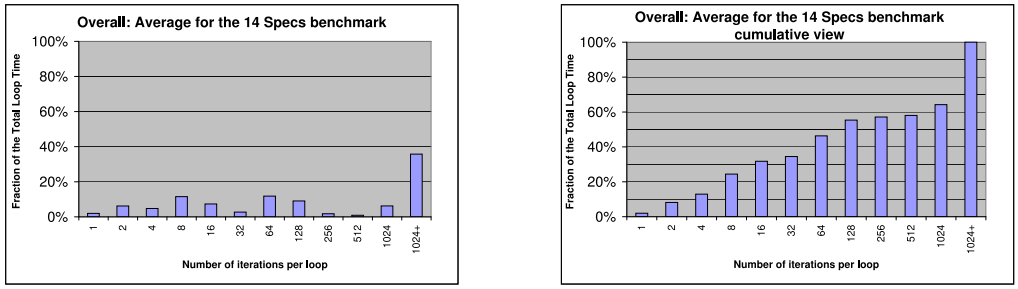


Figure 4.2: Displaying Hotpath information in MAQAO for 183.equake CG and a sample of its CFG.



(a) SPEC FP overall loop distribution

(b) SPEC FP overall cumulative loop distribution

Figure 4.3: Loop Execution Time depending on loops number of iterations.

is specially important since low-level optimization are always targeting the asymptotic performance, neglecting all start-up effects as cold start. For a short trip, start-up cost can be the dominant one.

From a performance analysis point of view, these numbers are taking advantage of both time and value profiling. Time profiling allows to give a precise weight to all executed loops, therefore underscoring hotspot. Value profiling monitors the iteration count. Correlating this information provides the relevant metric: i.e. which hot loops are short. This a clear illustration of the interest of centralized approach for performance analysis.

Instruction Level Profiling Knowing dispersion of values manipulated by each instruction can be valuable to take optimization related decision. For instance, characterizing address streams allows to detect bank conflict or aliasing problem. Observing that a given value is almost always constant can justify the cost of high level code specialization.

For a prefetch distance estimation, the algorithm is straightforward: all the instructions within the loop are instrumented. Data are stored in order sensitive way (rotating buffer). Evaluation of the prefetch distance can also be done statically based on a dependence graph (DDG). However, at the opposite of the static module, instruction instrumentation handles data stream interleaving (an optimization used Itanium by *icc* [53] to fetch alternatively two data streams with a single prefetch instruction)

A more comprehensive explanation on the techniques used in MAQAO, with some results on case studies can be found in [46, 48].

Hierarchical Reporting

A classic pitfall for reporting tools is to overload end-user under a mountain of data. Therefore, this trend leads to miss the initial goal which is to help in the decision process. The needs of the user differ, depending on which level the decision is going to be made: is it to chose between two compilers ? To select different compilation flags for the whole application ? To tune specifically a given loop? Being aware of this, MAQAO organizes information hierarchically. Each level of the hierarchy is suitable for a given level of decision to be taken: complete loop characterization, loop performance analysis, function analysis or whole code analysis. Additionally a filter can be set depending on the degree of confidence of the answer or the potential performance gain involved.

- The first and most exhaustive level is the instructions level view. For each loop, selected instructions counts and built-in metrics are displayed. These counts require some architecture knowledge to be interpreted but they represent the exact and complete input of what MAQAO is going to process in the upper stages. Instructions are coalesced per family (such as integer arithmetic, load instructions and so on) and counted on a per basic block basis. However the goal is not to catch dispersion rules (hence the taxonomy) of the architecture, but to detail instructions that have been determined as being of special interest. This instruction count is enriched by built-in metrics : Cycle cost per iteration, issue cost per iteration, Theoretical cycle bounds per iteration. Together counts and metrics are exploited by deductive rules that give the interpretations and possible optimization opportunities. Rules also process results gathered during application execution (instrumentation, hardware counter, cycle count).

- The second level, which is already an abstraction layer, only reports loop where some important performance features are detected, thus filtering out a large amount of non-essential data. Additionally results are reported in a user-friendly way (not just a bunch of number but clear sentences)

- The third and fourth levels are summary tables respectively for each routine and for the whole code. A report counting the number of detected performance issues is issued. Reading these tables is quick and was designed to ease comparison.

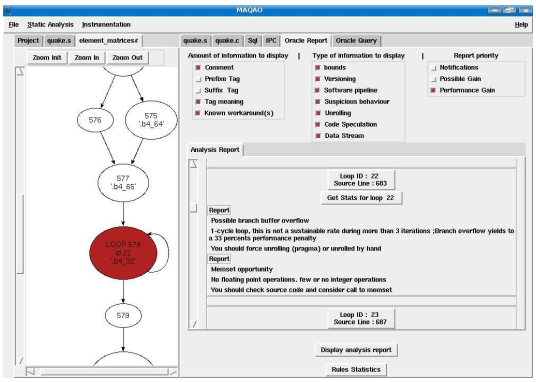
One of the key features of MAQAO is the possibility to express new analysis using scripts. This offers a wider range of analysis when standard statistics are not enough. Therefore, a knowledge base of interesting analysis can be built up from the experience of multiple users or from micro-benchmarking techniques[83]. The results of micro-benchmarks are patterns of code that do not perform well due to possible data alignment issues, conflicting memory banks or other memory access flaws. Scripting, in Lua [80], allows to experiment and tune new analysis with ease, and extends the tool to the advantage of other users.

The results of these reports can be used either interactively by the developer of the application, or automatically. Indeed, evaluation of the static performance helps in comparing codes that are generated by search strategy (as the one described in the previous chapter for instance), avoiding measure of the execution time for each code. MAQAO reports could guide an automatic exploration of optimization sequences or parameters, and for instance control unrolling factors (unroll until quality of the assembly code generated degrades due to spill/fill). Such as feed-back from assembly code to source optimizations, is similar to what was achieved in OCEANS project [1].

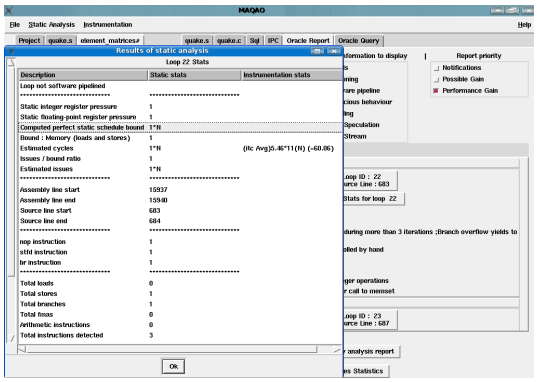
4.3 Assembly Specialization Opportunities

In this section, we study how specialization impacts compiler generated code, in theory from the source code point of view and in practice. The objective is to propose new techniques exhibiting performance of specialized codes while limiting code size expansion and performance overhead, taking advantage of compiler behavior.

Consider the following code and assume that $s > 0$:



(a) Loop selected in the CFG: Performance Advisor displays analysis. The loop is an one-cycle loop and is similar to a memset routine. Thus it should be replaced by a call to the hopefully optimized memset routine



(b) Analysis of the loop with inclusion of dynamic information. Performance is bounded by Memory. While the perfect static schedule is estimated to 1 cycle per Iteration instrumentation results show that the average loop trip count is 11 and the cost 5.5 cycle per iteration. Static estimation is exceeded by a factor of 5.

```
void f(double *A, int n, int s, int d)
 int i;
 for (i = 0; i <= n; i += s)
   A[i] = A[i-d]*2 ;
```

Concerning dependences on array `A`, there are three cases: Either there are PC loop-carried dependences ($d > 0, d\%s = 0$ and $d \leq s * \frac{n}{s}$), and the distance is d/s, or there are CP loop-carried dependences ($d < 0, -d\%s = 0$ and $-d \leq s * \frac{n}{s}$), with a distance of d/s, or there are no dependences, for all other cases. As far as dependences are concerned, these three cases can lead to three different code versions, each corresponding to a different schedule.

Now let us consider the corresponding codes generated by a compiler. When compiling using Intel ICC v9 compiler on an Itanium platform (see 4.5 for complete description) with `-O2` flag, the assembly loop generated is software pipelined and uses speculation. The code speculates that the dependence distance is greater than 5. In case of a correct prediction, the code takes 2 cycles/iteration. Otherwise, it takes more than 9 cycles/iterations.

There are several reasons why a production compiler does not generate different codes according to the conditions resulting from the dependence analysis:

- Computing dependence conditions as computed before requires a dependence analysis that is relatively expensive in terms of complexity, this kind of dependence analysis is usually not implemented in production compilers. Note that in some cases, the exact dependence conditions are out of range of any dependence analysis (when constraints are non linear for instance).

- The execution of the code evaluating the different conditions ($d > 0, d\%s = 0$ and $d \leq s * \frac{n}{s}$ for instance) would take valuable execution time. This time is considered as a runtime overhead and the compiler may choose to generate simpler code without this overhead.

If we add in the source three versions, one governed by the condition $d < 0, d\%s = 0$ and $d \leq s * \frac{n}{s}$, one with $d < 0, -d\%s = 0$ and $-d \leq s * \frac{n}{s}$, and the other taken when the two previous ones are not taken, the compiler does not improve the code generation: all three loops are the same, even if in one case, the loop is parallel.

We specialize the values d and s and observe the performance of the compiled loop for different dependence distances. We first study on the impact of the specialization of d and s, and then discuss the impact of specialization of n and s at the end of the section. Performance is measured in terms of cycle/iteration, and

```
void smvp(int nodes, ...)
  ...
  for (i = 0; i < nodes; i++)
    Anext = Aindex[i];
    Alast = Aindex[i + 1];
    sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
    sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][1][2]*v[i][2];
    sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + A[Anext][2][2]*v[i][2];
    Anext++;
    ...
```

Figure 4.4: Code fragment of `183.equake` benchmark

for each case the software pipeline depth, related to dependence distance is given.

PC Dependence distance	cycles/iteration	pipeline depth
≥ 4	2	6
$= 2, 3$	4	3
$= 1$	9	2
≤ 0	1	15
no dependence	1 or 2	15 or 6

When there is no dependence, some values of s and d generate code that takes 2 cycles/iterations, some others 1 cycle/iteration. Note that when the distance increases, the pipeline depth increases, leading to better performance, till the maximum of 1 cycle/iteration. All these versions reach a higher level of performance than the initial code. Is it possible to generate a code with the same performance, working with any value of s and d?

A static specialization would generate one code for all different values of s and d in a given range. If the range is large, this is clearly not a solution since the code size will exceed instruction cache size, causing extra cache misses. However, for different values of s and d, for instance $s = 1, n = d$ and $s = 1, d = 8$, the assembly codes generated are very similar: Only a few constants change, corresponding to the values of s and d. We will say that both versions have the same templates. Therefore, if we can dynamically change these values in a template according to the values of s and d, we would need only 4 versions of templates. The number of templates is related here to the different schedules the compiler can generate.

We considered only the specializations due to different dependences. There are other reasons to specialize a code according to some integer variable:

- Instruction selection: instructions may be valid only for some range of immediate values. For large values, other instructions may be required. For instance, post-increment of address registers is usually limited to small values.

- Loop trip count: loop trip count impacts prefetch distance in the case of regular code. Moreover, small loop trip counts give the opportunity for a full unroll for instance.

Consider the code in Figure 4.4 of the most time-consuming function *smvp* from `183.equake` benchmark. We generate all versions obtained by specializing the parameter `nodes` with constants from 1 to 8192. We define a relation among these versions: two versions are in relation if their object code differ only by immediate operands of some assembly instructions. This relation defines a notion of *classes* between versions. Versions in a same class have been optimized similarly by the compiler, the differences are only due to the different values for `nodes`. Analyzing the object codes generated by the Intel compiler *icc V9* corresponding to the 8192 specialized versions, we find 31 different *classes* of code. Figure 4.5 shows the classes obtained after specialization together with the number of versions in each class. Any version in the class can serve as a *template* which can be instantiated at runtime for many values. Such behavior of compilers is similar for other benchmarks as well, even for different architectures.

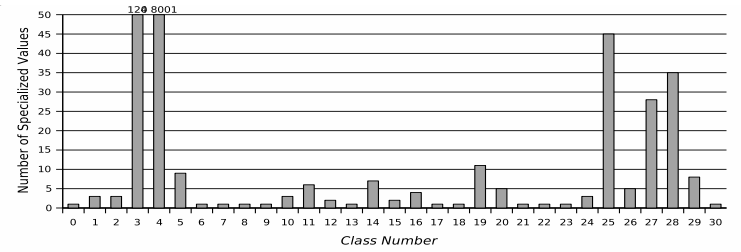


Figure 4.5: *Classes* obtained for *183.equake* benchmark

The principle of the optimization we propose in Section 4.5 relies on the fact that while versioning functions for different parameter values, the compiler does not generate completely different codes. For some parameter value range, these codes have the same instructions and only differ by some constants. The value range to consider can be defined by several approaches: profiling, user-input, or static analysis. The idea is to build a binary template, which if instantiated with the parameter values, is equivalent to the versioned code. If the template can be computed at compile time, the instantiation can be performed at run-time with little overhead. We therefore have the best of versioning and dynamic specialization, i.e., we take advantage of complex static compiler optimizations and yet obtain the performance of versioned code without paying the cost of code expansion.

Now, consider again the first motivating example. We specialize s to 1 and d to -1 and consider only the effects of specialization on the iteration count through the value of n. When n is parametric, the compiler assumes n is large enough and generates a pipelined loop with up to 15 stages. Moreover, the loop has some prefetch instructions, with a distance implying that at least 200 iterations are executed, otherwise the data prefetched is not used inside the loop (responsible for cache trashing). The compiler does not change the resulting assembly code when n is specialized for large values, 1000 for instance. For values of n that are smaller than 200, the compiler generates a code that performs better than the original code (we had to resort to a pragma in order to force the compiler to remove the prefetch instructions). For very small values of n, the compiler fully unrolls the loop, and the resulting code outperforms the code when n is not specialized. The traditional approach of code versioning, applied to the iteration count, would generate a decision tree that according to the value of n, and select a version unrolled, or with or without prefetches. A traditional multi-versioning approach would create a decision tree selecting either of these three cases and for each execution of the loop, only one of the version is run. We propose in Section 4.4 a new technique to combine all these versions into a code that switches smoothly from one to the other while the iteration count increases. The method is applied directly on the assembly codes and its distinctive feature is its use of a performance model, in order to predict when to switch from one version to the other (including the overhead taken by the switch itself).

4.4 Compositional Loop Specialization

Replacing input variables with constants enables a compiler to generate more optimized code. The idea of the approach is, given different codes (and schedules) for the same loop, to combine or compose them into one code that achieves the same performance as the best code, for any iteration count. It is understated that the quality of the resulting multi-versioned code will depend on the quality of the different initial versions. One of the difficulty of generating multi-versioned code is to predict when to switch from one version to the other. This requires a performance model, and we choose to analyze the assembly code in order to take into account the very precise instructions that will be executed. This performance model is detailed in the following section.

The analysis of the SPEC inner loops in Figure 4.3 shows that short loops matter for application performance. Half the time spent in inner loops is spent in loops with less than 128 iterations. Other applications

such as x264 (MPEG-AVC encoder) [140] or BLAST (genomic application) have been studied likely and they spend a large portion of their execution time in short loops. Generating good performance code requires different optimizations depending on the loop iteration count. Many optimizations are beneficial only on a given range of iterations. For each of the following optimizations, we describe its limitations and the conditions for which it applies

1. *peeling*; Peeling enables a rescheduling of the first iterations of a loop. It generates more opportunities for a better resource usage, with a free schedule, at the expense of code expansion.

2. *unrolling*; Unrolling a loop body offers the opportunity for better ILP. The higher the unrolling factor, the higher the impact on performance of the tail code for small loops.

3. *data prefetching*; Data prefetching cuts by a large amount the read/write latency of memory accesses. Tuning the prefetch distance is highly dependent on the total number of iterations. For small loops, the prefetch distance is too large for the prefetching to be effective. In this case, removing the prefetches may free resources for a better ILP, therefore increasing performance.

4. *software pipeline*; The Initiation Interval (II) is usually the value minimized by software pipeline algorithms, and represents the amount of time between two successive start of iterations. This comes at the expense of the latency required to execute a complete iteration (MAKESPAN), which is important for small loops.

This shows that there are many opportunities in which it would be interesting to combine different optimized codes according to the iteration count. The full details of the method presented here can be found in [45].

Performance Model

Consider two different optimized versions of the same loop, called L_1 and L_2. This can be generalized to any number of versions. We assume that these loops are inner loops (they do not include other loops). The cycle count of L_1 is given by the formula: $c_1(i) = \alpha_1.i + \beta_1$, where α_1 is an rational number, β_1 an integers and the cycle count is rounded down. Similarly for L_2, $c_2(i) = \alpha_2.i + \beta_2$.

We consider the case where the two loops are such that: $\alpha_2 < \alpha_1$ and $\beta_1 < \beta_2$, meaning that L_1 is faster than L_2 when $i < \frac{\beta 2 - \beta 1}{\alpha_1 - \alpha_2}$ and L_2 outperforms L_1 for larger number of iterations. We would like to build a *best* code such that:

$$\forall n, c_{best}(n) = \min_k(c_k(n)).$$

This *best* code is built by an optimization function min: $\min(L_1, L_2) = best$. This function min defines a minimum on codes w.r.t. performance, for all iteration values. Due to the difficulty of building the minimum of two codes without introducing any overhead, we propose to tackle a more pragmatic problem. We want to build a code $\min(L_1, L_2)$ with a level of performance very close to the performance of the best of the two codes. The following constraints are applied to the loop to build:

1. Asymptotic performance (in cycle/iteration, when iteration count grows) is the same than the best asymptotic performance of L_1 and L_2.

2. Each loop is possibly called many times, each time with a possible different loop trip count. For a given distribution of loop counts, the average gain in cycle/iteration compared to the best asymptotic performance of L_1 and L_2 is positive.

3. When performance of both loops L_1 and L_2 meet, the loop built moves to the best code for asymptotic performance.

Note that the second constraint does not compel the new loop to outperform L_2 and L_1 for each iteration count, but in general, for all the execution of the loop, some cycles have been gained. The reason is that

some overhead may appear when switching from one version to the other. The best code would have the following cycle count:

$$\mathrm{c}_{12}(i) \quad = \quad \begin{cases} i \leq B: & \mathrm{c}_1(i) \\ i > B: & \mathrm{c}_2(i) - \mathrm{c}_2(B) + \mathrm{c}_1(B) + \gamma \end{cases}$$

where B is the integer $\frac{\beta 2 - \beta 1}{\alpha_1 - \alpha_2}$ and γ represents the overhead necessary when going from one version to the other. This overhead represents register initializations, branch mispredicts,...

The difference in cycles/iteration between the asymptotic best loop and the new loop $\min(L_1, L_2)$ is, for an iteration count i:

$$dpi(i) = \begin{cases} i \leq B: & (\mathrm{c}_2(i) - \mathrm{c}_1(i))/i \\ i > B: & (\mathrm{c}_2(B) - \mathrm{c}_1(B) - \gamma)/i \end{cases}$$

The difference in cycle/iteration is asymptotically 0, meaning that this new version is as fast as L_2. When the loop iteration count is uniformly distributed among iterations $[1..N]$, the average difference in cycle/iteration is obtained by:

$$adpi(N) = \sum_{i=1}^{N} \frac{dpi(i)}{N}.$$

This definition can easily be adapted to other distributions. In particular, distribution of values caught during profiled execution can be used. When $adpi(N)$ is positive, it means that for a uniform distribution of loop trip counts in $[1, N]$, the new loop $\min(L_1, L_2)$ is in average faster than the best asymptotic loop L_2. This value is positive when $N < B$ since each difference/iteration is positive for all iterations $i < B$. For higher values of N, $adpi(N) > 0$ if:

$$\gamma < \frac{B(\alpha_2 - \alpha_1)(1 + H(N) - H(B)) + (\beta_2 - \beta_1)H(N)}{H(N) - H(B)}, \tag{4.1}$$

where $H(N)$ is the harmonic number $H(N) = \sum_{k=1}^{N} \frac{1}{k}$. As H is a strictly increasing function, this implies that when N asymptotically grows, γ must be such that:

$$\gamma < \mathrm{c}_2(B) - \mathrm{c}_1(B).$$

This constraint implies that the new loop $\min(L_1, L_2)$ takes less cycles than the best asymptotic version, for any value of the iteration count.

From this constraint we can deduce the basic steps to build the code $\min(L_1, L_2)$:

1. Compare $\mathrm{c}(L_1)$ and $\mathrm{c}(L_2)$, in order to compute B
2. Assuming L_1 outperforms L_2 for the first B iterations, evaluate the code in-between necessary for the transition and the overhead β generated.
3. If inequality 4.1 is satisfied, then build the minimum of the two codes. Otherwise the overhead is too significant with respect to the total execution time.

Among the limit of out method: the code is expanded. As with all other versioning schemes code size is traded for performance. If the number of iteration remains constant or at least in a single range the extra code size and some instruction overhead will penalize the execution time. However if we consider the Spec benchmark as representative of the average code complexity it be can safely stated that iteration range is varying a lot and that specialization on iteration number will mostly increases performance.

Compile-Time Assembly to Assembly Transformation

We first discuss on assembly code dependence analysis then describe two particular transformations, loop peeling and transformation of prefetching, as well as their composition. These two steps are for an Itanium

architecture, but we believe this can be generalized to other platforms. Such post-compiler optimization is already a hot topic of research [40, 112, 95].

In order to preserve code semantics, the validity of the transformations applied is checked by computing a data dependence graph (DDG) on the assembly codes. For memory accesses, the base rule is that all memory accesses are interdependent (read or write with write). Then, if the compiler schedules two instructions within less cycles than the minimum latency of the first instruction (the one being the possible dependency anchor), as the schedule is correct, then the two instructions are independent.

Peeling is the process of 'taking off' a number of iterations from a loop body, and consequently explicitly express them at the beginning or end of the loop. This is often done in order to match two different bounds of two subsequent loops. Generally, the positive effect of this technique is better understood if explained in conjunction with loop fusion. In our approach, the peeling has also a positive effect if explained in conjunction with software pipelining. Compared to warming up stages of software pipelined loop, an interleaving scheme does not increase latency but increases the number of iterations simultaneously in-flight. This does not yield to excessive register pressure. In fact, the global register pressure depends on the number of iterations simultaneously alive. Our peeling techniques is careful enough to keep this number below the software pipelined loop asymptotic behavior. The initial schedule of peeled iterations is the schedule obtained after a possible flattening. Then iterations are jammed (or interleaved) with a list scheduling algorithm with priority to the first iterations. The statements of the first iteration peeled are scheduled first and have higher priority over the statements of the second iteration. Indeed, this schedule improves over the initial schedule, w.r.t. the difference in cycle per iteration, as presented in Section 4.4. Finally, a mechanism is needed to ensure program correctness if the number of total iterations is smaller than the number of peeled iterations. One calculated branch is used for a late entry into the peeled code, and predicate registers guard interleaved instructions, preventing interleaved instructions that do not belong to the desired peeled iterations from being commit. Thus the overhead is limited to a couple of cycles, taking into account computation of the predicates and computation of the branch register.

For prefetching, the prefetch distance is estimated from the computation of addresses (relative initial values, increment). We then assess the number n of first loop iterations that do not take advantage of the prefetch. The loop is split into a sequence of two similar loops. The first loop has no prefetches (they are replaced by nops) and has n iterations. The second loop is the initial loop, performing the remaining iterations.

Experimental Results

We consider three benchmarks: a `DAXPY` loop ($Y[i] = \alpha \times X[i] + Y[i]$) and two benchmarks from the SPECs: `GALGEL` and `MGRID`. The `DAXPY` illustrates the combination of both unrolling and prefetch specialization.

`DAXPY:` Prefetch instructions must be generated for both `X` and `Y` arrays. It appears, that using prefetch degrades the initiation interval of the software pipelined loop due to extra pressure on memory slot. Based on our performance model, there are three versions of the initial code:

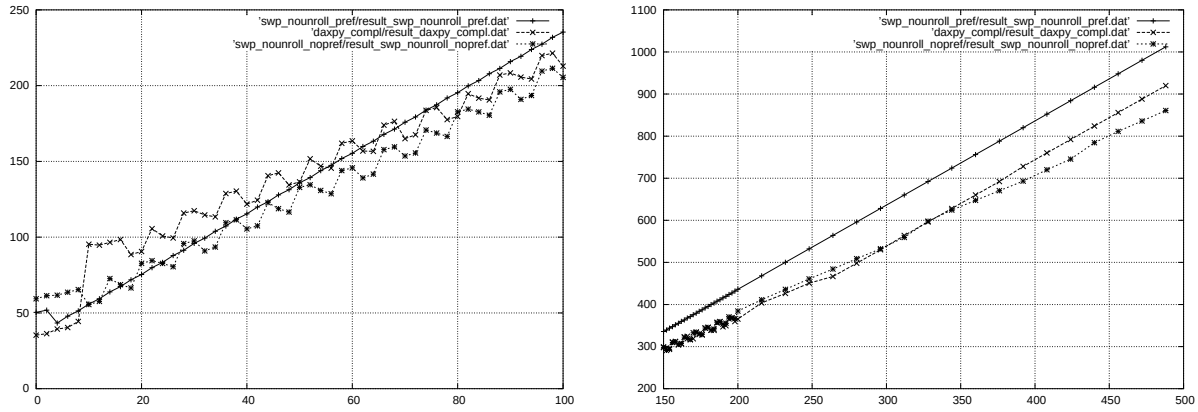
- First zone: peeling, each block of 8 iterations costs $30 + N \bmod 9$. Peeling degree is set to 8 since is corresponds to the minimal latency in the DDG (4 cycles) which is just enough to schedule 8 floating point instructions.
- Second zone: disable prefetch, the formula is in $1 \times N + \alpha$[1]
- Third zone: enable prefetch, formula: $2 \times N + \alpha$

From the prefetch version, we know that the prefetch distance is set to 800B. Therefore, considering that every iteration is consuming 8 Bytes, it means that the loop needs to iterate at least 100 times before accessing the first prefetched data. These 100 first iterations can be generated without prefetch.

Performance results are detailed in Figure 4.6. Figure 4.6(a) is a close-up of the relative performance between composed versioning, prefetch and no prefetch. It appears that for the first 8 iterations, composed

[1] The Itanium architecture can not sustain one branch per cycle without inserting a stall cycles, the real formula is $1.7 \times N + \alpha$

versioning outperforms all the other versions. However for the 9^{th} iteration, composed versioning suffers from the overhead of filling up pipeline, while they are already filled-up for both other versions. This is consistent with our policy: overheads should be postponed as far as possible. Therefore even if these overheads still account for the same number of cycles their *relative cost* is smaller. Notice, that clearly composed versioning follows the same behavior than no prefetch version. In this example we chose to stick with no pretech up to 128 iterations. Figure 4.6(b) details the behavior for large number of iterations. Composed versioning sticks with the no prefetch slope outperforming the prefetch version up to a hundred iterations. Beyond, it sticks to the original version, which is the best asymptotic version compared to the no-prefetch version.



(a) Short iteration count for composed versions

(b) Asymptotic behavior for composed versions

Figure 4.6: Performance measured between original software pipelined loop, original without prefetch, original with peeling and prefetch turned off for the first 128 iterations.

`GALGEL, loop b1_20:` The loop *b1_20* is a pipelined loop, and is one of the many versions generated by the compiler for one source loop. The loop has 8 iterations in the `train` input data set, 11 in the `ref` input data set. For this loop, we performed a peeling of one iteration. Table 4.7 sums up the results of the peeling transformation.

Iteration count	Cycles		Performance Model		Gain
	Orig.	Peeling	Orig.	Peeling	
8	37760118	33984118	16xN+32	16xN+28	2.5 %
11	373744918	344995318	16xN+32	16xN+28	1.92 %

Figure 4.7: Static analysis of code with peeling of one iteration out of loop *b1_20*

`MGRID, loop b7_81:` The loop *b7_81* in the assembly code is memory access intensive, since it performs in two cycles two load-pairs (equivalent to four loads) and four stores. The loop uses one prefetch instruction and is pipelined. Peeling the loop does not bring significant performance gain, according to the performance model. Indeed each peeled iteration takes 2 cycles and interleaving peeled iterations does not reduce this latency. Therefore, as soon as the loop trip count exceeds the number of iterations peeled off the loop, the cycle count of the optimized loop should be similar to the cycle count of original loop.

As for prefetching, we split the loop into a sequence of two similar loops, the first without any prefetch instruction. The histogram of loop trip counts, provided by MAQAO and presented in Figure 4.8 shows that the loop trip counts are small enough to make prefetch useless. Indeed, by removing prefetches in this single loop, the performance gain obtained for the whole benchmark is 25%. This illustrates a case where prefetches are counter-productive and trashes the data cache.

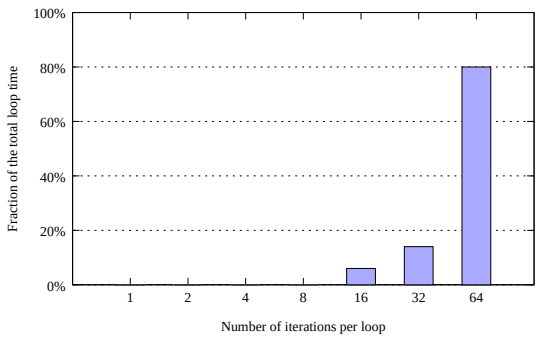


Figure 4.8: `MGRID` loop b7_81 execution time depending on number of iterations. The histogram summarizes iteration weights for an interval in power of two: $]0,1],]1,2],]2,4],]4,8]$ and so on.

4.5 Hybrid Specialization

The runtime specialization of values is also achieved by dynamic compilation systems [124, 71, 122, 98, 72, 99, 111] and off-line partial evaluators [37, 39] but they involve time expensive activities such as code generation with memory allocation. Thus these systems require hundreds of calls to amortize the overhead but still need to keep different versions during execution (one version for single specializing value).

For the hybrid specialization approach summarized here (a full description, joint work with M.A. Khan and H.-P. Charles can be found in [87, 88]), we do not require such heavyweight activities. Specialization is performed for a limited number of instructions in a generic binary template. This template is generated during static compilation and is highly optimized since we expose some of unknown values in the source code to the compiler. The template is then adapted to new values during execution thereby avoiding code explosion as in other existing specializers. The versioning therefore needs to be performed only for those cases where the template generation was not possible. We have applied our method to different applications and libraries including `FFTW3` [64], `ATLAS` [156] and SPEC CPU2000 [76].

Principle of Hybrid Specialization

Code specialization can be accomplished either at static compile-time (versioning) or at run-time, with a dynamic code specializer. On the one hand, versioning, or function cloning results in performance improvement results from computation simplifications (through constant propagation, dead code elimination), or by triggering other complex optimizations such as software pipelining. On the other hand, run-time optimizations take advantage of the knowledge of input and function parameter values to perform further transformations. The drawback for the former is the code explosion due to large domain of input values, and for the latter, the optimization time is an overhead to the application execution time. Optimizations that take time but may not bring enough performance to compensate the optimization overhead are therefore not appropriate at run-time.

The principle of the optimization we propose relies on the fact that while versioning functions for many different parameter values, the compiler does not generate completely different codes. For some parameter value range, these codes have the same instructions and only differ by some constants. The value range to consider can be defined by several approaches: profiling, user-input, or static analysis [43]. The idea is to build a binary template, which if instantiated with the parameter values, is equivalent to the versioned code. If the template can be computed at compile time, the instantiation can be performed at run-time with little overhead if the binary template is simple enough to instantiate. We therefore have the best of versioning and dynamic specialization, i.e., we take advantage of complex static compiler optimizations and yet obtain the performance of versioned code without paying the cost of code expansion.

We define the notion of template as an abstraction of binary codes: a template is a binary code with some slots (positions in the binary) corresponding to parameters of assembly instructions. These slots can

be filled with the constant values instantiating the template. Let $T_{X_1 \ldots X_n}$ denote a binary template with slots $X_1, \ldots, X_n$. The instantiation of this template with the constant integer values $v_1, \ldots, v_n$ is written $T_{X_1 \ldots X_n}[X_1/v_1, \ldots, X_n/v_n]$, and corresponds to a binary code where all slots in the template have been filled with the values. The instantiation of a template has a very low complexity, and relocatable codes can be considered as examples of binary templates.

Now consider the code of a function F to be optimized, we assume without loss of generality that F takes only one integer parameter X. This function is compiled into a binary function $C(F)(X)$, where C denotes the optimization sequence and code generation performed by the compiler. By versioning F with a value v, the compiler generates a binary $B_v = C(F_v)$, performing better than $C(F)(v)$. We define an equivalence between specialized binaries:

Definition 2 (Binary template) *Given two specialized binaries B_v and $B_{v'}$, B_v is equivalent to $B_{v'}$ if there exists a template $T_{X_1 \ldots X_n}$ and functions $f_1 \ldots f_n$ such that*

$$T_{X_1 \ldots X_n}[X_1/f_1(v), \ldots, X_n/f_n(v)] = B_v, \quad T_{X_1 \ldots X_n}[X_1/f_1(v'), \ldots, X_n/f_n(v')] = B_{v'}.$$

In other words, the two specialized binaries are equivalent if they are instantiation of the same template with the same function applied to their specialized value. Let R denote this equivalence.

We have shown that this relation is indeed an equivalence.

Computing the minimum number of templates necessary to account for all specialized binaries B_v when $v \in [lb, ub]$ boils down to compute the equivalence classes $\{B_v, v \in [lb, ub]\}/\mathcal{R}$ incrementally. The following graph shows the relation between specialized binaries and templates:

Interval of values		**Specialized binaries**		**Binary templates**
$[lb, ub]$	$\longrightarrow$	$\{B_v, v \in [lb, ub]\}$	$\rightleftharpoons$	$\{B_v, v \in [lb, ub]\}/\mathcal{R}$

As shown in the motivating example of Section 4.3, there are much more specialized binaries than binary templates. Given the range of values to specialize for, compilation of the specialized binaries from the original code is achieved by a static compiler. Computation of the templates is likewise at static compile time. Instantiation of the templates corresponds to the dynamic specialization, at run-time. As this is only a substitution and application of a function to v, the cost of the dynamic substitution depends on the number of slots in the template and cost of the functions used.

As we are interested in a lightweight dynamic specializer, taking as little overhead as possible, we restrict the template to use only affine functions. It means that the functions $f_1, \ldots, f_n$ involved in the instantiation of a template must be of the form: $f_i(x) = \alpha_i.x + \beta_i$. It can be shown that affine templates no longer define an equivalence class. However, it is still possible to compute how many specialized code versions correspond to a given affine template.

Main Steps

We describe in this section the main steps leading to the optimized hybrid code, incorporating both static and dynamic specializations. These steps, as exposed in Figure 4.9 are:

1. Profiling and selection of most frequently used parameter values together with their intervals for hot functions: We require to instrument the application code to register the hot regions of code and apply specialization only to such regions/functions. The code is instrumented to also perform value profiling [26] for integer parameters that a function obtains. The intervals of values are then used to proceed for dynamic specialization whereas a limited set of values in each interval is used for static specialization.

2. Versioning of selected functions; We generate different specialized versions of the same function, where one of its integer parameters is replaced by a constant. These versions will be used in both static specialization (for most frequently used values) and dynamic specialization (for intervals).

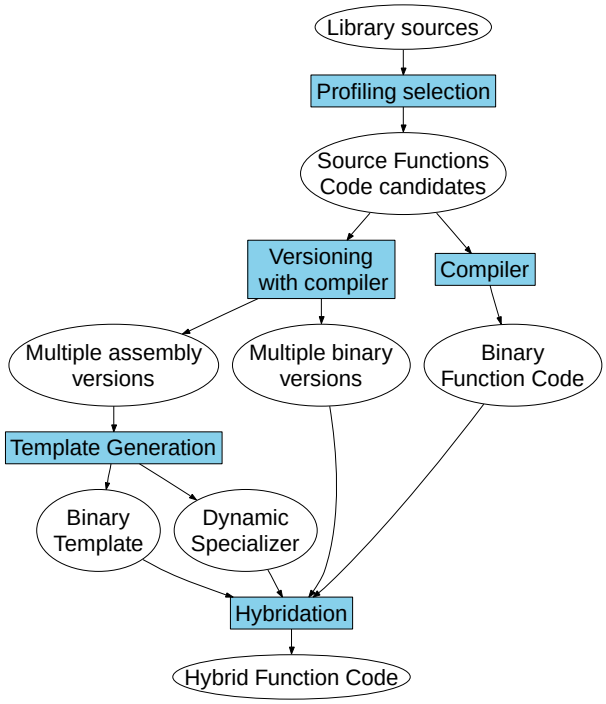


Figure 4.9: Overview of Hybrid Specialization

3. Analysis of assembly code within intervals for template creation; To proceed for dynamic specialization, we need to generate one specializer and single template for each interval. Therefore multiple versions of assembly code are searched within one interval.

4. Using the versions found in previous step, generation of runtime specializer (if possible); If the system of equations can not be solved i.e. n consistent versions are not found, then dynamic specialization does not proceed further, since no generic template could be found. In case, where the equations are solved and constants α and β are found for each difference in all the n versions, then specializer generation takes place together with generation of binary template. Subsequently, the information regarding starting location, offset of binary instruction to modify and new value is gathered. In order to modify binary instructions during execution, the self-modifying code can be generated based upon the information gathered. We make use of a lightweight runtime Instruction Specializer that accomplishes the task of binary code modification (only p binary instructions) in an efficient manner.

5. Hybridization, i.e., putting statically specialized versions, the template code with the dynamic specializer (if generated through last step) and the initial code altogether for the hybrid version; To limit number of specialized versions for a function, we take into account the number of valid templates found. So this number of templates will actually reduce the number of static versions. Moreover, the source code is modified by inserting necessary code for redirection to specialized code versions.

The hybrid specialization approach (depicted in Figure 4.9) has been automated for function parameters of integral data types. These parameters should be defined through a directive of the form :

#xlang specialize *parameter_name*[, *parameter_name*..]

The prototype can perform various activities including: parsing profiling information obtained through MAQAO [47], analysis of assembly code versions, finding differences among these versions followed by generation of runtime specializer if possible, and the hybridization of the versions according to the specified criteria. In this section, we describe details of each of these steps over Itanium-II architecture.

Processor	Speed	Compilers
Intel Itanium-II (IA64)	1.5 GHz	gcc v 4.3, icc v 9.1
Intel Pentium-4 (R)	3.20 GHz	gcc v 4.3 , icc v 8.0

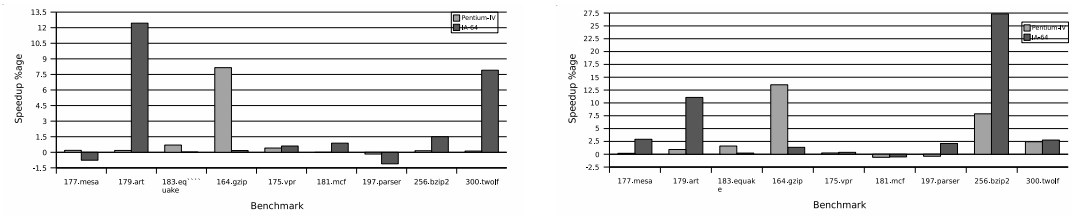
Figure 4.10: Configuration of the Architectures Used

Experimental Results

This section summarizes the results of hybrid specialization after being applied to `SPEC CPU2000` and `FFTW3` benchmarks. The experiments have been performed over the platforms with the following configurations in Figure 4.5.

Itanium 2 processor and Intel compiler icc v9.0, using `libpfm` to measure the execution speed.

Different benchmarks in CPU2000 suite have been optimized using hybrid specialization with reference inputs. The speedup %age obtained w.r.t standard code has been shown in Figure 4.11. For `SPEC` benchmarks, the speedup is not large, because in many cases the hot code does not always include integer parameters or the candidate parameters are unable to impact overall execution to a large factor.



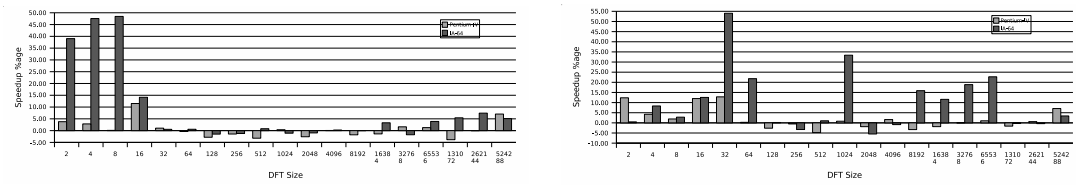
(a) Speedup %age with *icc* compiler (b) Speedup %age with *gcc* compiler

Figure 4.11: Performance Results of `SPEC CPU2000` Benchmarks

For benchmark *mesa*, the compilers are able to perform inlining and partial evaluation. However, it does not have any significant impact on large portions of code.

In *art* benchmark, the most important optimization for IA64 is data prefetching and unrolling For *equake*, the large values of specializing parameters resulted in code almost similar to that of unspecialized with small difference in code scheduling. The *gzip* benchmark takes advantage of loop optimizations and code inlining for Pentium4. However, for IA64, the compilers generated similar codes. In the *vpr*, *mcf* and *parser* benchmarks, most hot functions do not use integer parameters and the large range of in runtime values reduces the performance gain after hybrid specialization.

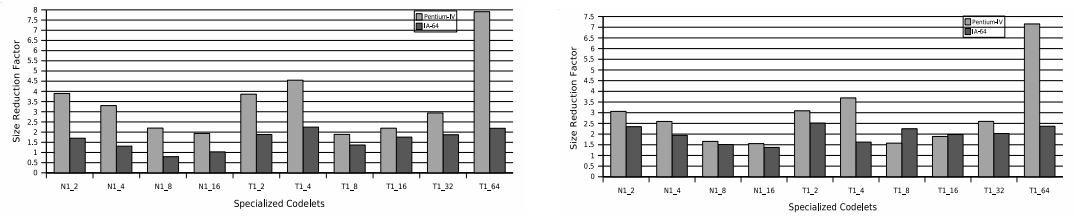
`FFTW` library contains C routines to compute Discrete Fourier Transform (DFT) of real and complex data and of arbitrary input size in *O(nlogn)*. It incorporates FFTW *wisdom* which is the best configuration for the corresponding architecture which can be generated before execution (off-line) to reduce search time during execution. Figure 4.12 shows the speedup obtained for calculating complex DFTs of powers of 2. The calculation of single DFT may comprise repeated invocations of multiple codelets.



(a) Speedup %age with *icc* compiler (b) Speedup %age with *gcc* compiler

Figure 4.12: Performance Results of `FFTW`

It is clear that the main potential to achieve speed-up through specialization exists for the smaller and the larger values of DFT size. For medium range values there are two reasons explaining the lack of speed-up: the codelets selected by *FFTW wisdom* are not the best suitable candidates for specialization. For instance, `m1_64` is a loop with a large body and it limits the ability of the compiler to take advantage of specialized values. The number of calls to the specialized codelet is also too small to impact the overall application execution time by a large factor. The FFTW library also provides codelets which are already specialized with stride values (of powers of 2) and may be used to improve the performance of its kernels. The figure 4.13 shows the *Reduction Factor* (Size of already specialized Codelets/Size after hybrid specialization) for these codelets. The code with large radix has more code size, and therefore hybrid specialization achieves large *Reduction Factor*.

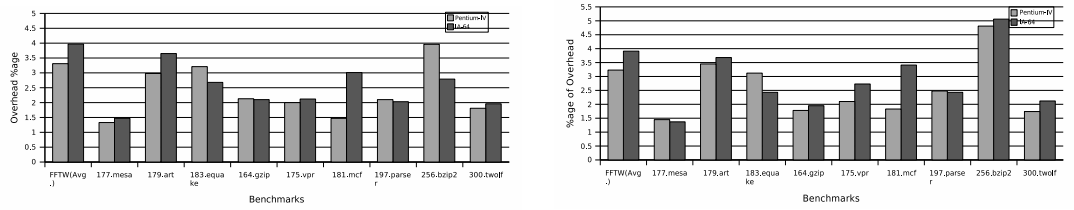


(a) *icc* compiler (b) *gcc* compiler

Figure 4.13: *Reduction Factor* obtained for different codelets

A summarized view of overhead with respect to application execution time is shown in Figure 4.14. It is to be noted that overhead of runtime specialization is very small since the modification of single instruction takes average[2] of 9 cycles on Itanium-II and 2 cycles on Pentium-IV.

The calculation of offset of specialized data is performed once per runtime specialization of entire template followed by cache coherence(required for IA-64). On Pentium-IV, the cost of runtime specialization is small, however the costs of function call and branches which are larger than those of Itanium-II, make the entire overhead almost equivalent to that in Itanium-II.



(a) *icc* compiler (b) *gcc* compiler

Figure 4.14: Overhead of Specialization

4.6 Related Works

The related works are organized in two parts: instrumentation and performance analysis tools and specialization techniques, either static or dynamic.

[2]The binary instruction formats require extraction of different bit-sets

Instrumentation and Performance Analysis Tools

Two main classes of low level instrumentation tools can be related to MAQAO. One class consists of performance analysis tools aiming at understanding application behavior based on hardware counters. Fall in this category tools such as VTune (self-contained program), or PAPI (user-independent). The other family of tools is more focused on code manipulation like Salto, or code instrumentation such as ATOM or PIN. However MAQAO broad approach is more related to the path chosen by HPCview or to Finesse, the later being more focused on parallelization than code optimization.

Hardware counters are extremely helpful for performance tuning, they are the backbone of tools such as `VTune`[153], `Caliper` [78] or Kojak [162]. Their usage is so widespread that an API gets standardized to describe their access [23]. Nevertheless, hardware counters are limited to the description of the dynamic behavior of an application and this picture needs to be correlated with other metrics. For instance, from the hardware counter point of view, code bloating (or even dead code) filling up functional units and leading to high IPC is seen as a desirable behavior.

On the static side, `Salto` [131] is a framework dedicated to the implementation of complex assembly code transformations. The assembly is first parsed and then it is seen as a collection of C++ objects plugged into a user-developed application. `Salto` is more a toolkit than a tool and could appear as a back-end of MAQAO diagnosis chain: once a problem is identified by MAQAO, some transformations have to be applied by SALTO to solve this problem. DPCL [132] (Dynamic Probe Class Library) is a set of C++ classes from IBM originally based on Dyninst [24]. The purpose is to help developers to support dynamic instrumentation of parallel jobs. Probes can be inserted in a running binary to check the hardware counters or cycles for any function of the monitored code. Even if dynamic instrumentation is very appealing, DPCL does not include any notion of code inspection.

`ATOM` [138] (for Alpha processors) and `Pin` (for Intel processors) [120] instrument assembly codes (or even binary for `Pin`) in a way that instrumented specific instructions trigger the execution of user-defined routines. `Pin` can perform instrumentation at the function level, block level or instruction level. In particular, value profiling of registers (address registers for instance) is possible. However, there seems to be no global analysis of the collected values, necessary for the hot-path computation and for the computation of prefetched array sections. Moreover, `Pin` does not collect profile information at the loop level and does not correlate information gathered with static information. This implies that identifying a compiler optimization that does not fit the input data (such as deep software pipeline or prefetching for a loop with a small iteration count) is not possible with `Pin`, whereas MAQAO enables this kinds of analysis. EEL (Executable Editing Library) [95] belongs to the same categories of tools. This C++ library allows to edit a binary and add code fragment on edges of application CFG. Therefore it can be used as a foundation for an analysis tool but does not provide performance analysis by itself. Currently EEL is available on SPARC processors. Periscope [65] also belongs to the same category, and is a tool focused on the run-time detection of performance issues. These issues are specified by conditions and properties on measures captured by monitors. Run-time performance analysis is achieved in a distributed and on-line manner. While Periscope keeps information about the structure of the code (loops, functions), however there is no analytical evaluation of its performance and no comparison between statically predicted performance and performance measures.

`HPCview` [110] and `Finesse` [116] (this one being more oriented toward parallelization) address the analysis problem from static and dynamic sides. `HPCview` tackles the same problem as MAQAO: the complex interaction between source code, assembly, performance and hardware monitors. `HPCview` presents a well-designed GUI based on web browser, displaying simultaneous views of source, assembly code and dynamic information. This interface is connected to a database storing for each statement of the assembly code a summary of its dynamic information. Based on control flow graph and a tool named `bloop`, `HPCview` builds abstracted representation of code loop structures (using an XML interface). Some important differences should be underscored: while a database is embedded in the application, end-user has only limited opportunity to explore the code and define new queries. `HPCview` also lacks value profiling which can lead to powerful, yet simple to implement optimizations such as code versioning.

Shark [135] developed by Apple offers a comprehensive interface for performance problem. As MAQAO, it analyzes assembly code and displays source code as well as profiling information. However Shark lacks

instrumentation and value profiling, code structures are not displayed and the performance analysis delivers currently a limited number of messages concerning alignment, unrolling or Altivec usage (vectorization). Additionally Shark does not offer scripting language or standard database. Nevertheless it traces many dynamic behaviors (including call stack, garbage collection, binary analysis), and it underlines the need to think performance software beyond gprof.

Specialization Related Works

Specialization is a well known technique to obtain high performance programs, either statically or dynamically (with a dynamic specializer).

Static specialization is often used in conjunction with profiling: Using profiling information collected by precedent executions, the compiler generates optimized versions of loops or functions according to the most frequently used values of loop bounds or of parameters. These compiled time specialization often boils down to the generation of codes that are in mutually exclusive execution paths. Loop specialization resulting in index set splitting has been studied by Griebl *et al.*[73], for a different goal: their goal is to partition the iteration space according to the dependence pattern for each statement. This increases control but increases the number of affine schedules that can be computed for each code. Tiling is another transformation that splits the iteration domain into tiles for better scheduling (changing the cost of memory latency). However, very few works resort to loop versioning in order to explicitly reduce the overall latency of the loop. This is due to the intractability of general performance models. That is one reason why asymptotic loop counts are generally considered for optimization.

The optimizations involved in our specialization scheme – software pipelining, peeling and prefetching – are classical optimizations. Software pipelining[9] is a key optimization for VLIW and EPIC architectures. In particular, modulo scheduling, as used by the `ICC` compiler, exhibits instruction parallelism, even in the presence of data dependencies, that greatly improves performance. Modulo scheduling targets large iteration counts and tries to find an initiation interval (II) as small as possible, defining the throughput of the loop. However, when the iteration count is small, this may increase the loop latency (in particular when the II is essentially constrained by resources). Loop peeling is a well known technique for improving the behavior of the code for small iteration count. As it comes at the expense of code size, compiler heuristics usually prefer to not use it. With our approach, it is possible to decide, according to the awaited iteration count distribution, whether peeling is worth or not. Moreover, our technique would take advantage of profile information since the distribution is then more accurate. Finally, prefetch works by bringing in data from memory well before it is needed by a memory operation. This hides the cache miss latencies for data accesses and thus improves performance. Typically, when a cache line is brought in from memory, it contains several elements of an array that is accessed in the loop. This means that a new cache line is needed for this array access only once in several iterations. This depends on several factors such as the stride of the array access pattern, the cache line size, etc. In fact, if redundant prefetch instructions are issued every iteration, it may degrade performance. Prefetch instructions require extra issue slots and thereby increase the cycle per iteration ratio. Redundant prefetches can overload the memory subsystem and thereby adversely affect the performance, and prefetch too much in advance can also result in cache pollution and performance degradation [53]. Prefetches are interesting only when the iteration trip count is large enough to make data access at the prefetch distance. This implies that for medium iteration numbers, prefetch instructions can be removed.

Dynamic specialization is an active field, in particular for interpreted languages with Just In Time compilers.

Tempo [39, 117] is a specializer that can perform compile-time and runtime specialization. A detailed comparison has been given in Table 4.1. The static compile-time activity by Tempo includes partial evaluation that is only applicable when the values are static (i.e. already known). In hybrid specialization, we expose the unknown values to generate template. Therefore the template is more specialized in our case than the one generated through Tempo specializer. In normal programs where most of the values are dynamic, the overhead of specialization will increase to a huge factor thus degrading the performance of the application. However, in the hybrid specialization approach, only calculation of specializing values is required since we are

	Hybrid specialization	Tempo	Tick C
	Compile Time		
Approach	Declarative	Declarative	Imperative
Analysis Type	Static Analysis + Invariants Analysis	Binding + Evaluation time Analysis	Static Analysis
Template Type	Single generic template	A large number of templates for single function	Intermediate form generated by lcc
Synthesis Type	Specializer generation	Specializer generation	Code generator generation
	Dynamic View		
Memory Allocation and Code Copy	No Dynamic Memory Allocation or Copy	Memory Allocation and Code Copy to buffer	Memory allocation and Code Copy to buffer
Dynamic Computation	Calculation of only specializing values	Evaluate all expressions in the code	optimizations and register management
Branch Transformation	No Branch Relocation	Branches Relocation due to code movement	No branch relocation
Cache Handling	Cache coherence required	Cache Coherence required	Cache coherence will be required
Code Generation Level	Instruction-wise Binary Code Specialization	Complete buffer generation and specialization	Misc. activities including code generation, register allocation, Live variable analysis, environment binding etc.
	Optimizations		
Implemented Optimizer	Benefits from all aggressive optimizations by exposing value at static compile time + dynamic specialization	Partial evaluation + runtime optimizations by Tick C	With VCODE, dynamic unrolling + strength reduction + constant folding + dead-code elimination
	Runtime Overhead		
Computations at Runtime	Calculation of value to specialize binary instruction	If program has no static values, then entire evaluation of code is dynamic	Heavy code generation and optimizations
Generation Time	12 to 20 CPI[3]	Same as Tick C	100 CPI if ICODE otherwise 300 to 800 CPI with VCODE

Table 4.1: Comparison of Main Activities Related to Runtime Specialization and/or Code Generation

specializing a small number of binary instructions. Moreover, runtime activities other than optimizations, such as code buffer allocation and copy, incur large amount of overhead thereby making it suitable for code to be called multiple times.

C-Mix [104] is a specializer that is able to perform partial evaluation only at static compile-time. It can specialize the code by propagating information up-to branches and then generating different specialized constructs. However it is not suitable for the cases where large part of information is not available at static compile-time, which is normal for most of the real-life applications.

Tick C(‘C)[124] is a superset of ANSI C and uses ICODE and VCODE interfaces together with *lcc* retargetable intermediate representation to generate dynamic code. It is able to achieve large speedup by performing optimizations during execution of code. However, its code generation activity incurs a large amount of overhead requiring more than 100 calls to amortize the overhead. In case of hybrid specialization approach, we minimize the runtime overhead with generation of optimized templates at static compile time. Some other dynamic code generation systems have been suggested in [98, 101, 100] that categorize the compilation process into different stages at which different optimizations can be performed. These models are effective enough to improve performance, however they require the programmer intervention to decide which optimization could be appropriate at which stage. Similarly, DCG [56], CCG [122] and other approaches in [72, 71, 38] suggest efficient dynamic code generation and specialization systems, however these systems are different in that these can not be used to produce *generic* templates requiring large number of dynamic template versions for each different specializing value.

In runtime optimization systems, Dynamo [11] developed at HP Labs. is used to interpret instruction stream to optimize code during execution without requiring user annotations, instrumentation or off-line profiling information. It directly operates on the native code, searches for hot traces, optimizes them and then stores in a fragment cache for execution. Other frameworks such as Trident [167] and ADORE [103] make effective use of multiple threads to perform dynamic optimizations (e.g. cache prefetching, Value Specialization etc.) based on profiling information. The speedup is achieved at a low-cost, however they require continuous monitoring of hardware counters for processor events to capture the information necessary

[3]Cycles per generated instruction

to perform these optimizations.

4.7 Conclusion

Performance evaluation is essential for code optimization, and assembly codes are a mine of information for evaluation and for optimization tuning. Following the path pioneered by OCEANS project [1] and developed by iterative compilation (Bodin *et al.* [20], Kisuki *et al.*[90]), we have proposed techniques and a tool, MAQAO, to provide feed-back from assembly code analysis and instrumentation in order to guide the search for compiler options and source code optimizations. The combination of instrumentation, value-profiling, with static assembly code analysis delivers precise diagnoses for code tuning that are out of reach of a stand-alone instrumentation or simulation approach. The essential features are the ability to identify performance bottlenecks (or to identify a case where there are no performance problems), to compare performance of two versions statically, and to refine code performance analysis and guide further optimizations. Currently MAQAO analyzes and instruments codes generated by different compilers (`icc` and `gcc` on IA32 and IA64). The performance analysis is still under development for IA32 architecture (coupling static and dynamic results). Ports for other architectures, in particular for embedded architectures are planned for future works. More complex performance model are planned, in particular for caches, and for instruction decoding performance for IA32 architectures.

From a larger perspective, checking code quality is easier than generating code with high quality. Assessing the quality of compiler generated codes is essential but we assume that feed-back information can only guide semi-automatically the optimization process. While some hints delivered by tools like MAQAO can be integrated into an automatic compilation chain, either for source optimizations or for assembly to assembly optimizations, some hints need application and context knowledge to decide how to impact the source code or optimization sequence.

We have explored two techniques for assembly to assembly optimizations. Both techniques aim the generation of multi-versioned code generation based on static/dynamic analysis of compiler-generated assembly codes, a compositional approach for loop specialization and an hybrid specialization for some cases of integer variable specialization. Hybrid specialization builds multiple templates created statically from compiler-generated codes and instantiated dynamically with a low overhead. Both methods are based on specialization and rely on the fact that only a limited number of versions are able to improve performance significantly, for a wide range of values.

The stem of our work on compositional approach to loop versioning is the diagnosis that a consequent fraction of execution time is spent in loops with a small number of iterations. However, even modern compilers seem to bet everything on asymptotic performance. Clearly there are performance opportunities for non-asymptotic behaviors and optimization must be adapted to the size of data, and for loop, to the iteration range. This compositional versioning limits the overhead, reduces costly decision tree height and exploits and executes as much as possible of the generated code. This new technique is based on loop versioning, according to the iteration count distribution. This is a generalization of simple asymptotic evaluations. Given a loop count distribution, either coming from static analysis of the code, provided by the user through pragmas, or observed by profiling, we propose a smart loop versioning scheme. In particular, we split index sets so that each iteration range can be optimized more aggressively. The proposed optimizations are, for short range: peeling and for medium range: turning prefetching off, in addition to any versions proposed by the compiler. The first results on SPEC benchmarks show up to 25% speed up for one benchmark.

Hybrid specialization achieves dynamic specialization on top of static specialization. From the similarity existing between codes specialized for many different values and optimized by a compiler, a limited number of templates time is created. These templates are then instantiated dynamically, by a lightweight specializer. The resulting code has therefore performance of a statically specialized code, for many different values, with a limited code size expansion. This method gives good speedups compared to the original code, even for highly optimized code such as FFTW and SPEC benchmarks over different platforms and compilers. The method essentially rely on the capacity of the compiler to generate specialized versions that improve performance compared to the non-specialized code. An important aspect of this automated hybrid method is the low

overhead of runtime specialization. Since, the template is generated at static compile time, it is highly optimized, and during execution, only a limited number of binary instructions is then specialized. Moreover, the hybrid approach provides a solution to the code expansion issue caused by versioning and constitutes an interesting alternative to usual tradeoff between library size and performance achieved.

Chapter 5

Perspectives

Perspectives of the work presented in this document are organized along three themes: performance models and their role in adaptive compilation framework, the place of libraries for the optimization and generation of high performance applications, and extensions and perspectives of this thesis to parallelism.

5.1 Performance Modeling

Performance modeling is essential for code optimization. The increasing complexity of hardware however hampers analytical performance prediction and makes performance models inaccurate. The formalization of the phase ordering problem presented in Chapter 1, deciding how to organize sequences of optimizations, shows that this problem is untractable as soon as models are inaccurate. From a practical point of view, hardware mechanisms such as caches or prefetches generate threshold effects and discontinuity in performance w.r.t. input data that are difficult to model. As a consequence, measuring performance by running applications is necessary since it is the only reliable performance evaluation, but at the same time, performance measure for an input data set does not help much to predict performance for another input data set.

Tools such as MAQAO presented in Chapter 4, VTUNE [153] or HPCView [110] help developers to optimize their application by analyzing performance measures and assembly code. This category of tools now belongs to the toolbox of any developer of high performance application. MAQAO for instance proposes feed-back based on comparison between performance measures and performance predicted by an analytical model and delivers hints for compiler flags, pragmas or source code transformations. Automatic expertise of code quality, through static and/or dynamic analysis, relies on a knowledge basis of the architecture and of the compiler. Micro-benchmarks reveal the behavior of some hardware mechanisms, measure precise latencies and may exhibit compiler shortcomings. X-Ray [166] and lmBench [109] are micro-benchmark suites measuring architecture parameters common to different architectures. For more detailed architecture behaviors, hand-tuned benchmarks, such as those proposed by Hager *et al.* [74], Jalby *et al.* [83] or Agner [2] in his very precise optimization manuals, are designed to measure the latencies of some specific instruction or hardware mechanism. The automatic generation of micro-benchmarks for new architectures and compilers is still an open issue and is one of the challenges for having better performance models and tuning tools.

Analytical performance models are still very useful and cannot be replaced completely by empirical search, even if modeling all hardware behaviors seems out of reach and may not be desirable anyway. We believe instead that micro-benchmarks and expertises built from their results should help to design new models that abstract away behaviors that are not important for performance. In other words, a better understanding of performance mechanisms is necessary to build better (but not more complex) performance models. For instance, for many years, decreasing the number of cache misses was synonym of improving performance. The miss penalty was so important that performance models could focus only on counting cache misses. As shown in the introduction of Chapter 3, this is no longer true for Intel (IA32 and IA64) architecture

for instance and new abstractions (simplifications) have to be found. We have shown by using hierarchical compilation that a trade-off between ILP and locality enabled better performance than minimizing cache misses. Similarly, at a time where a synchronization between two cores can be as fast as a cache access, trade-offs are probably to be adapted to parallelism (and memory bandwidth usage). The hierarchical approach proposed in this document shows one possible direction for further investigation: Performance model for hierarchical compilation only considers code decomposed into coarse grain kernels while kernel performance are measured. The effects of hardware mechanisms operating on a small time window, such as out of order execution, automatic prefetch, dispersal rules, ..., can be neglected at a coarse grain level of execution, whereas the kernel performance measures take them into account. The model only focuses on the effects of caches and we have shown that performance prediction is very accurate. We then used this performance prediction for the generation of multi-versioned library codes.

Performance evaluation is particularly important in the context of library generation. Library functions must have good performance for all input values (in particular, input sizes), and multi-versioned codes are generally the appropriate solution. Iterative compilation techniques (initiated by the works of Bodin *et al.*[20] and Kisuki *et al.*[90]) explore optimization parameters and optimization sequences, using empirical search. Many active libraries resort to this principle: Atlas[156] for linear algebra, Spiral [129] and FFTW [64] for signal processing, Phipac [19] for sparse linear algebra, Stapl [147] for standard templates and for sort, the self-tuning sorting library proposed by Li *et al.*[102]. Selection of the best version according to input values (or sizes) may also be achieved through experimental search on training sets. Empirically driven techniques, based on machine learning methods [130, 114], are used for instance in Stapl [148], and they can be combined with analytical models containing implicit algorithmic knowledge (as for FFTW or Spiral). In [66], Fursin *et al.* propose to apply iterative compilation during one execution (instead of many), comparing performance of different versions of hot functions when execution enters a stable phase. The benefits are of course a reduced execution time but also the ability to generate a multi-versioned code that selects the best version adapted to the different execution phases (instead of input values as in other techniques). We proposed in Chapter 3 to generate multi-versioned codes with an analytical model built on top of experimental measures of kernel latencies. While empirical models and machine learning techniques may drive efficiently the selection of the best version in a given context, we believe only analytical models are able to drive the exploration from a higher level, potentially bypassing execution phases. Important reduction of exploration time would widen the range of applicability of adaptive compilation.

5.2 Libraries and Optimizations

High performance libraries can be defined as a collection of functions, independent of any application context, that deliver performance for any valid input data. Refactoring an application with highly optimized libraries is usually considered as an engineering issue, and basically libraries are used in order to reuse correct code, gain productivity and obtain performance. The number of library functions is however limited by what the developer can handle and these functions correspond to well-known algorithms. As a matter of fact, decomposing an algorithm using libraries for code reuse is a different issue from using libraries for performance. The intuitive decomposition, from an algorithmic point of view, may not be the best from a performance perspective.

We believe high performance libraries, combined to semantic properties of operators could be the building blocks of application optimizations. Assume we have an application written using library functions, for the purpose of improving developer productivity. We first have to restructure the code so that large code fragments match operations of a specific application domain (such as SPL operators[163] for instance). These code fragments are then expressed in a higher level representation or formula, abstracting away syntactic details and artifacts of the language. Using semantic knowledge, the formula is optimized, considering the application context, and the optimized code is generated from the new formula expression. While this approach may seem still far-fetched, there are a number of works showing the path. Note first that this approach is a divide-and-conquer approach for code optimization: implicitly, we assume that optimizing an application can be achieved by piece-wise optimization. As shown by Schwiegelshohn *et al.*[134], notion of

best performance code has no meaning, we assume that this divide-and-conquer method is able to generate codes with good performance instead, as defined in Chapter 1, Section 1.2. For the above approach to work, three sub-problems have to be addressed:

- Define an abstraction or formula language for which optimization is simpler,
- Given an application, find code fragments or patterns that fit in this abstraction, eventually applying transformations to ease detection,
- Optimize and generate efficient functions for these code fragments, taking into account the application context. Libraries are then aggregates of functions optimized by this method.

We consider in turn each of these subproblems, which are interesting by themselves and are tackled by many papers.

Following the principle that the more information given to the compiler, the better the code generated, optimization is simpler on codes with simple iteration domains or codes corresponding to computations for which there is a richer semantics. Consider codes with constant loop bounds in hierarchical compilation for instance, they overcome the untractability of the optimization problem for general iteration domains (Theorem 3, Chapter 1) since optimizing basic blocks for multiple issue architectures is decidable[1]. Hybrid compilation in Chapter 4 also gives more information to the compiler by specializing read-only variables into constants. Multiple specialized versions are then 'generalized' back through dynamic templates. There are other abstractions, more elaborate, that use richer semantics. The main benefit of these frameworks or abstractions is not exactly to simplify optimization, but to make possible semantically-based optimizations, potentially generating much better codes. Solar-Lezama *et al.*[136] propose a method for the optimization of combinatorial computations using semantics of arithmetics and boolean functions, under some conditions. SPL [163] is a representation for signal processing computations (used in Spiral), that uses a rich semantics on the linear algebra SPL operators. This enables Spiral to explore many different formulations semantically equivalent to a given function. The polyhedral model or its extension, Z-polyhedral model [68], represents static control parts [18] of an application. The polyhedral model corresponds to a data-flow representation of the program, abstracting away control and dependences coming from variable reuse. A program in its polyhedral representation boils down to the expression of a mathematical definition of recurrence. Gautam and Rajopadhye [67] define semantic rules for the reductions in the polyhedral model and their method is able to optimize the computational/space complexity of programs. All these representations are algorithm-independent, even if they can represent only some kind of computations. They are therefore appropriate for piece-wise optimization of general applications, provided the operators they use, conveying rich semantics, are recognized in the application.

Methods for the recognition of pre-defined functions and templates have been proposed in Chapter 2. The recognition does not need a restructuring step and variations coming from control transformations (such as fusion, fission, interchange, skewing...) or data-layout transformations are transparent to the method. This works well for the detection of predefined sets of operators, and the limited SPL operator detection shown in the conclusion of Chapter 2 is a promising application to extend. As a matter of fact, automatic recognition of functions or patterns offers the opportunity to have in libraries more functions than what the programmer can handle. In particular, algorithmic variations of a code could be coped with by considering detection of the most frequent variations (This technique is used in Chapter 2 to some extent). Other methods rely on a restructuration step in order to isolate the code that matches conditions for optimization. Static control parts, for a polyhedral representation, are built by explicit code transformations, such as constant propagation, induction variable detection, ... Likewise, in Chapter 3, an exploration of transformations, guided by X-language pragmas, is used to define constant performance kernels. How to guide these transformations in order to obtain interesting code fragments is left for future work.

High performance code generation is difficult, in part due to the fact that performance models do not capture the complexity of real machines and performance evaluation is achieved through execution of generated codes. For library generation, compilation time is off-line and iterative techniques can be considered.

[1] as a matter of fact, NP-complete

They are the backbone of active libraries, and Pouchet *et al.*[126] have proposed techniques adapting iterative compilation for the polyhedral model. For more general applications, methods based on experimental search must only consider a small number of parameters or input data sets (searching for compiler flags for instance) or put a high confidence on machine learning techniques[2]. The method that we are interested in assumes that all performance-critical computations of applications can be written as library functions or templates (or sequences of functions, optimized contextually). Optimizing the application then boils down to optimizing library code.

The dwarves defined by Asanovic *et al.*[137] correspond to algorithmic methods that capture patterns of computation and communication. They do not define a precise library interface but equivalence classes where membership in a class is defined by similarity in computation and data movement. These 13 dwarves cover a large range of application domains, and the underlying assumption is that all applications resort to algorithms or patterns that are in dwarves. The approach we propose to adopt would help to optimize applications by using algorithms/templates belonging to dwarves. If libraries of functions and templates are the key, an open question is whether all computations that match dwarves correspond to a finite number of functions/templates, and what is this number. While the theoretical answers would be 'no' and 'infinite', it could be interesting to determine these answers from a more practical point of view.

5.3 Parallelism

This document has focused on compiler-centric problems and Chapters 3 and 4 have tackled compilation issues for monocore architectures. Extensions to multi-core architectures and interactions with the operating system are the main focus of our future work.

Parallelism begins by its detection or expression in a code. A quick overview of the main categories (or dwarves) given in [10] shows that only a small number of them corresponds naturally to regular codes. Assuming dwarves are representative of computation and communication patterns that are used in most applications, automatic detection of parallelism in general requires precise data dependence analysis on irregular codes. This analysis must be able to deal with non regular accesses or control. The instance-wise data-flow dependence analysis proposed in our PhD thesis [12] some years ago belongs to the analyses proposed in the literature for irregular codes (along with methods proposed by Pugh and Wonnacott [128], Creusillet and Irigoin [44] or Rus, Rauchwerger and Hoeflinger [133] for instance). An open-source implementation done by Marouane Belaoucha is in-progress and its integration in GRAPHITE [125], a polyhedral representation for gcc, will follow[3]. The key feature of this method is that it relies on the polyhedral model, where non-linear properties are attached to some parameters. A very interesting problem not yet explored remains scheduling and code generation from the information produced by this method. In particular, a transformation like deep jam [29] corresponds (form a scheduling point of view) to an unroll and jam for while loops. Describing deep jam in the polyhedral model and automatically checking its legality would show the way for more general code transformations on non-regular codes. Automatic correction of transformations on irregular codes, as proposed for regular codes by Vasilache *et al.* [152], would be a great asset for meta-compilation languages such as X-language.

Parallelism for multi-cores involves run-time support. Running threads, placing threads on physical cores, and memory management are generally left to the operating system. These elements are essential to prevent a number of issues among which load balancing and false sharing. Affinity of threads for a particular core[149], the one whose cache contains the required data, or affinity between threads, sharing some common data (see Marcel thread library and BubbleSched scheduler [144] for instance) are notions that the compiler can find and pass to the dynamic scheduler of the OS. Affinity between threads is a property on the threads coming either from user specification or from the compiler. The dynamic scheduler takes advantage of this information according to its knowledge of the architecture. Marcel is an adaptive library for threads, and it specializes itself with respect to the underlying architecture. If the architecture is hierarchical (SMT,

[2] After all, performance may be a one-way function of inputs that could not be captured by machine learning techniques if P≠NP.

[3] supported by TeraOps project [141]

ccNUMA), Marcel allows the application or compiler to specify different levels of affinity (called bubbles) and the scheduler of Marcel, BubbleSched takes advantage of the hierarchical architecture for the effective execution of threads. From a compiler perspective, this implies that without recompilation, the code can run efficiently on machines with different cores. We believe that such mechanism, passing compile-time information to the dynamic scheduler, is necessary to improve scalability of applications and could be a way to generalize the hierarchical compilation described in this document.

Finally, scalability of parallel applications is one of the main objectives to reach for the compilation on many-core architectures. Identifying performance bottlenecks and being able to predict how performance is going to change when the data size or the number of cores increases are important steps for the design of scalable applications. Performance analysis tools for parallel codes, such as Kojak [162] focus on the automatic detection of performance bottlenecks for parallel languages (MPI, OpenMP and SHMEM for Kojak). Coupling inter- and intra-thread performance analysis, by integrating MAQAO and Kojak, is one of the goal of the European project PARMA [119]. The challenge here is to integrate tightly inter- and intra-thread performance evaluations: a high performance thread code may use most of the memory bandwidth, which is detrimental in multi-threaded environment. Tuning parameters (compiler flags, pragmas) for the compilation of threads must therefore be adapted accordingly. More generally, performance trade-offs, trading individual thread performance for scalability for instance, has to be explored.

Bibliography

[1] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. R. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P. M. W. Knijnenburg, M. F. P. O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E. A. Stohr, M. Verhoeven, and H. A. G. Wijshoff. Oceans: Optimizing compilers for embedded applications. In *Euro-Par Conference*, Lect. Notes in Computer Science. Springer-Verlag, 1997. 15, 58, 74

[2] F. Agner. Optimization manuals. `http://www.agner.org/optimize`. 77

[3] T. Alexander. Performance Prediction for Loop Restructuring Optimization. Master thesis, University of Carnegie Mellon. Physics/Computer Science Department, July 1993. 10

[4] C. Alias. *Program Optimization by Template Recognition and Replacement*. PhD thesis, Université de Versailles St Quentin, December 2005. 19, 25

[5] C. Alias. Tema: an efficient tool to find high-performance library patterns in source code. In *Workshop on Patterns in High Performance Computing*, Urbana-Champaign, May 2005. 29, 31

[6] C. Alias and D. Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *IEEE Working Conf. on Reverse Engineering*, pages 296–305, Victoria, Nov. 2003. IEEE. 25

[7] C. Alias and D. Barthou. Deciding where to call performance libraries. In *Euro-Par Conference*, volume 3648 of *Lect. Notes in Computer Science*, pages 336–345, Lisboa, Sept. 2005. Springer-Verlag. 22, 25

[8] C. Alias and D. Barthou. On Domain Specific Languages Re-Engineering. In *ACM Int. Conf. on Generative Programming and Component Engineering*, volume 3676 of *Lect. Notes in Computer Science*, pages 63–77, Tallinn, Sept. 2005. Springer-Verlag. 22, 26

[9] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995. 72

[10] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006. 8, 80

[11] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000. 73

[12] D. Barthou. *Analyse du flot des données pour tableaux en présence de contraintes non affines*. PhD thesis, Université de Versailles, Versailles, Feb. 1998. 80

[13] D. Barthou, C. Bastoul, F. Bodin, A. Cohen, L. Djoudi, C. Eisenbeis, P. Lesnicki, Z. Liu, O. Pène, S. Pop, and L.-N. Pouchet. Rapport d'analyse de performances et restructuration d'algorithmes de la chromodynamique quantique sur réseau (lqcd). ANR project PARA: Deliverable T5.4, July 2007. 47

[14] D. Barthou and P. Carribault. Optimization report for ci-1 and ci-2 codes. ANR project PARA: Deliverable T7.1, May 2007. 46

[15] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Symp. on Principles of Programming Languages*, pages 98–106, San Diego, California, Jan. 1998. ACM Press. 38

[16] D. Barthou, S. Donadio, A. Duchateau, P. Carribault, and W. Jalby. Loop optimization using adaptive compilation and kernel decomposition. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, pages 170–184, San Jose, California, Mar. 2007. IEEE Computer Society. 44

[17] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *Euro-Par Conference*, volume 2400 of *Lect. Notes in Computer Science*, pages 309–313, Paderborn, Aug. 2002. Springer-Verlag. 19, 21, 25

[18] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 23–30, College Station, Texas, Oct. 2003. Springer-Verlag. 79

[19] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *ACM Int. Conf. on Supercomputing*, July 1997. 78

[20] F. Bodin, T. Kisuk, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feed-back directed Compilation*, 1998. 74, 78

[21] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Int. Conf. on Computer-Aided Verification*, volume 818 of *Lect. Notes in Computer Science*, pages 55–67. Springer-Verlag, 1994. 25

[22] E. L. Boyd, G. A. Abandah, H.-H. Lee, and E. S. Davidson. Modeling computation and communication performance of parallel scientific applications: A case study of the ibm sp2. In *ACM Int. Conf. on Supercomputing*, 1995. 56

[23] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The Intl. Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000. 71

[24] B. R. Buck and J. K. Hollingsworth. An api for runtime code patching. *J. of High Performance Computing Application*, 14:317–329, 1994. 71

[25] C. Caβcaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *ACM Int. Conf. on Supercomputing*, pages 150–159, San Francisco, CA, 2003. ACM Press. 49

[26] B. Calder, P. Feller, and A. Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, 1997. 54, 56, 67

[27] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *J. of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988. 10

[28] Caps entreprise. `http://www.caps-entreprise.com`. 51

[29] P. Carribault, A. Cohen, and W. Jalby. Deep Jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, St-Louis, Missouri, Sept. 2005. IEEE Computer Society. 47, 80

[30] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, pages 111–122, 2005. 35

[31] A. Cimetile, A. D. Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3):145–178, 1996. 28

[32] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, pages 278–295. ACM Press, 1996. 26, 38, 42

[33] C. Click and K. D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995. 10

[34] A. Cohen, S. Girbal, and O. Temam. A Polyhedral Approach to Ease the Composition of Program Transformations. In *Euro-Par Conference*, Aug. 2004. 11

[35] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press. 37

[36] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1997. release October, 1rst 2002. 22

[37] C. Consel and O. Danvy. Tutorial notes on partial evaluation. *In ACM Symposium on Principles of Programming Languages, Pages 493-501, 1993*, 1993. 66

[38] C. Consel, L. Hornof, François Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation. International Seminar.*, pages 54–72, Dagstuhl Castle, Germany, 12-16 1996. Springer-Verlag, Berlin, Germany. 73

[39] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: Specializing Systems Applications and Beyond. *ACM Computing Surveys*, 30(3es), 1998. 66, 72

[40] K. D. Cooper, A. Dasgupta, and K. Kennedy. Vizer: A system to vectorize intel x86 binaries. In *Intl. Symp. on Computer Architecture*, Santa Fe, NM, 2002. 64

[41] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22, 2002. 11

[42] K. D. Cooper and T. Waterman. Investigating Adaptive Compilation using the MIPSPro Compiler. In *Los Alamos Computer Science Institute Symp.*, October 2003. 49

[43] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY. 66

[44] B. Creusillet and F. Irigoin. Exact versus approximate array region analyses. In *Int. Workshop on Languages and Compilers for Parallel Computing*, pages 86–100, 1996. 80

[45] L. Djoudi, J.-T. Acquaviva, and D. Barthou. Compositional approach applied to loop specialization. In *Euro-Par Conference*, volume 4641 of *Lect. Notes in Computer Science*, pages 268–279, Rennes, Aug. 2007. Springer-Verlag. 54, 62

[46] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. Exploring application performance: a new tool for a static/dynamic approach. In *Los Alamos Computer Science Institute Symp.*, Santa Fe, NM, Oct. 2005. 57

[47] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W.Jalby. Exploring application performance: a new tool for a static/dynamic approach. In *Proceedings of the 6th LACSI Symposium*, Santa Fe, NM, Oct. 2005. 68

[48] L. Djoudi, D. Barthou, O. Tomaz, A. Charif-Rubial, J.-T. Acquaviva, and W. Jalby. The design and architecture of maqaoprofile: an instrumentation maqao module. In *Workshop on Explicitly Parallel Instruction Computing Techniques*, San Jose, California, Mar. 2007. 57

[49] S. Donadio. *Optimisation itrative de bibliothque de calculs par division hirarchique de codes*. PhD thesis, Université de Versailles St Quentin, September 2007. 33

[50] 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 *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 4339 of *Lect. Notes in Computer Science*, Hawthorne, New York, Oct. 2005. Springer-Verlag. 35

[51] J. E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965. 22

[52] J. E. Doner. Tree acceptors and some of their applications. *Journal of Comput. and Syst. Sci.*, 4:406–451, 1970. 22

[53] G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing software data prefetches with rotating registers. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001. 57, 72

[54] E. Eckstein, O. Knig, and B. Scholz. Code instruction selection based on ssa-graph. In *Int. Workshop Software and Compilers for Embedded Systems*, volume 2826 of *Lect. Notes in Computer Science*, Vienna, Austria, September 2003. Springer-Verlag. 26

[55] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *J. of Instruction-Level Parallelism*, 5, 2003. Electronic journal : www.jilp.org. 10

[56] D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 263–272, San Jose, California, 1994. 73

[57] M. A. Ertl. Optimal code selection in DAGs. In *ACM Symp. on Principles of Programming Languages*, 1999. 26

[58] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. 42

[59] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991. 21, 24

[60] M. Fowler. *Refactoring, Improving the Design of Existing Code*. Addison-Wesley, 1999. 20

[61] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, page 221, Washington, DC, USA, 1999. IEEE Computer Society. 49

[62] 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. 10

[63] M. Frigo. A Fast Fourier Transform Compiler. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1999. 12

[64] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *IEEE Int. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998. 19, 66, 78

[65] K. Fürlinger and M. Gerndt. Peridot: Towards Automated Runtime Detection of Performance Bottlenecks. *High Performance Computing in Science and Engineering*, pages 193–202, 2005. 71

[66] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. *Transactions on High-Performance Embedded Architectures and Compilers*, 1(1):13–31, 2006. 78

[67] G. Gautam and S. Rajopadhye. Simplifying reductions. In *ACM Symp. on Principles of Programming Languages*, pages 30–41, Charleston, South Carolina, Jan. 2006. ACM. 79

[68] G. Gautam and S. Rajopadhye. The z-polyhedral model. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 237–248, New York, NY, USA, 2007. ACM. 79

[69] M. Gordon. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, San Jose, CA, Oct. 2006. 7

[70] K. Goto and R. van de Geijn. On reducing tlb misses in matrix multiplication. Technical report, The University of Texas at Austin, Department of Computer Sciences, 2002. 33, 49

[71] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-Directed Run-Time Specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 163–178. ACM, June 1997. 66, 73

[72] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Dyc : An expressive annotation-directed dynamic compiler for c. Technical report, Department of Computer Science and Engineering,University of Washington, 1999. 66, 73

[73] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000. 72

[74] G. Hager, T. Zeiser, J. Treibig, and G. Wellein. Optimizing performance on modern hpc systems: learning from simple kernel benchmarks. *Computational Science and High Performance Computing*, 91:273–287, 2006. 77

[75] N. Halbwachs, P. Caspi, D. Pilaud, and J. A. Plaice. LUSTRE / A declarative language for programming synchronous systems. *ACM Symp. on Principles of Programming Languages*, 215:178–188, 1967. 7

[76] J. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000. 56, 66

[77] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975. 24

[78] R. Hundt. Hp caliper: An architecture for performance analysis tools. In *Workshop on Industrial Experiences with Systems Software*, San Diego, CA, October 2000. `http://www.hp.com/go/caliper`. 71

[79] IBM. *Engineering and Scientific Subroutine Library, Guide and Reference, Release 3*, 4 edition, 1988. 33

[80] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua — an extensible extension language. *Software: Pratice and Experience*, 26(6):635–652, June 1996. `http://www.lua.org`. 58

[81] J. Jaeger and D. Barthou. Rapport d'analyse de performances et restructuration pour une application de modlisation molculaire. ANR project PARA: Deliverable T5.3, July 2007. 47

[82] 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. 10

[83] W. Jalby, C. Lemuet, and X. L. Pasteur. Wbtk: a new set of microbenchmarks to explore memory system performance for scientific computing. *Int. J. High Perform. Comput. Appl.*, 18(2):211–224, 2004. 39, 54, 58, 77

[84] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, 1996. URL: `http://www.cs.umd.edu/projects/omega/`. 42

[85] K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Int. Parallel and Distributed Processing Symposium (IPIPS'00)*, pages 297–304, 2000. 31

[86] 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. 10

[87] M. A. Khan, H.-P. Charles, and D. Barthou. An effective automated approach to specialization of code. In *Int. Workshop on Languages and Compilers for Parallel Computing*, Chicago, Illinois, Oct. 2007. 54, 66

[88] M. A. Khan, H.-P. Charles, and D. Barthou. Improving performance through low-overhead specialization of code. In *Int. Workshop on Iteraction between Compilers and Computer Architectures*, Phoenix, Arizona, Feb. 2007. 54, 66

[89] S.-M. Kim and J. H. Kim. A hybrid approach for program understanding based on graph-parsing and expectation-driven analysis. *Journal of Applied A.I.*, 12(6):521–546, Sept. 1998. 27, 28

[90] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *CPC*, pages 35–44, 2000. 74, 78

[91] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric Multi-level Blocking. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1997. 37

[92] I. Kodukula and K. Pingali. Transformations for imperfectly nested loops. In *ACM Int. Conf. on Supercomputing*, page 12, Washington, DC, USA, 1996. IEEE Computer Society. 37

[93] 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 *Conf. on Languages, Compilers, and Tools for Embedded Systems*, Washington, DC, June 2004. ACM. 10

[94] J. R. Larus. Whole program paths. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 259–269, 1999. 56

[95] J. R. Larus and E. Schnaar. Eel: Machine-independent executable editing. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1995. 64, 71

[96] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, Palo Alto, California, Mar 2004. 29, 31

[97] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, 1998. 38

[98] M. Leone and R. K. Dybvig. Dynamo : A staged compiler architecture for dynamic program optimization. Technical report, Indiana University, 1997. 66, 73

[99] M. Leone and P. Lee. Optimizing ml with run-time code generation. Technical report, School of Computer Science, Carnegie Mellon University, 1995. 66

[100] M. Leone and P. Lee. A Declarative Approach to Run-Time Code Generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996. 73

[101] M. Leone and P. Lee. Dynamic Specialization in the Fabius System. *ACM Computing Surveys*, 30(3es), 1998. 73

[102] X. Li, M. Garzaran, and D. Padua. A dynamically tuned sorting library. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, Palo Alto, CA, Mar. 2004. 78

[103] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. *Journal of Instruction-Level Parallelism*, 6, April 2004. 73

[104] H. Makholm. Specializing c- an introduction to the principles behind c-mix. Technical report, Computer Science Department, University of Copenhagen, June 1999. 73

[105] W. Mangione-Smith, T.-P. Shih, S. Abraham, and E. Davidson. Approaching a machine-application bound in delivered performance on scientifique code. In *Proc. of the IEEE*, volume 81, pages 1166–1178, Aug. 1993. 10

[106] B. D. Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Int. Workshop on Program Comprehension*, pages 164–174. IEEE Computer Society, 1996. 27, 28

[107] Y. Matiyasevich. Elimination of quantifiers from arithmetical formulas defining recursively enumerable sets. *Math. Comput. Simul.*, 67(1-2):125–133, 2004. 15

[108] D. Maydan, S. Amarasinghe, and M. Lam. Array dataflow analysis and its use in array privatization. In *ACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, SC, Jan. 1993. 24

[109] L. McVoy and C. Staelin. lmbench. `http://www.bitmover.com/lmbench/`. 77

[110] J. Mellor-Crummey, R. Fowler, and G. Marin. Hpcview: A tool for top-down analysis of node performance. In *Los Alamos Computer Science Institute Symp.*, Santa Fe, NM, October 2001. 71, 77

[111] S. Meloan. The java hotspot performance engine: An in-depth look. Technical report, Sun Microsystems, 1999. 66

[112] M. Merten and M. Thiems. An overview of the impact x86 binary reoptimization framework. Master's thesis, 1998. 64

[113] R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization.* MIT Press, 2000. 27, 29

[114] T. Mitchell. *Machine Learning.* McGraw Hill, 1997. 78

[115] Intel Math Kernel Library (Intel MKL). Intel. 33

[116] N. Mukherjee, G. Riley, and J. Gurd. FINESSE: A prototype feedback-guided performance enhancement system. In *IEEE Int. Conf. Parallel and Distributed Processing*, Rhodes, Greece, January 2000. 71

[117] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages (ICCL'98)*, February 1998. 72

[118] ANR Project PARA. http://www.projet-para.uvsq.fr. 46

[119] ITEA PARMA Project. http://www.parma-itea2.org. 81

[120] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium program s with dynamic instrumentation. In *ACM/IEEE Int. Symp. on Microarchitecture*, Portland, Oregon, 2004. 71

[121] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. on S.E.*, 20(6):463–475, June 1994. 28

[122] I. Piumarta. Ccg: Dynamic code generation for c and c++. Technical Report 25, INRIA Rocquencourt, october 2003. 66, 73

[123] G. Pokam and F. Bodin. An offline approach for whole-program paths analysis using suffix arrays. In *Int. Workshop on Languages and Compilers for Parallel Computing*, Lect. Notes in Computer Science, pages 363–378. Springer-Verlag, 2004. 56

[124] M. Poletto, W. C. Hsieh, D. R. Engler, and F. M. Kaashoek. 'c and tcc : A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21:324–369, March 1999. 37, 66, 73

[125] S. Pop. GRAPHITE: Loop optimizations based on polyhedral model for gcc. In *GCC Developer's Summit*, Ottawa, Canada, June 2006. 80

[126] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society. 80

[127] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In IEEE, editor, *ACM Int. Conf. on Supercomputing*, pages 4–13. IEEE Computer Society, 1991. 24

[128] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20(3):635–678, May 1998. 80

[129] M. Puschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. 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. 8, 12, 13, 19, 20, 78

[130] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. 78

[131] E. Rohou, F. Bodin, A. Seznec, G. L. Fol, F. Charot, and F. Raimbault. Salto : System for assembly-language transformation and optimization. Technical Report RR-2980, IRISA, 1996. 71

[132] L. D. Rose, T. Hoover, and J. K. Hollingsworth. The dynamic probe class library: An infrastructure for developing instrumentation for performance tools. In *IEEE Int. Conf. Parallel and Distributed Processing Symp.*, volume 66, 2001. http://www.ptools.org/projects/dpcl. 71

[133] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static and dynamic memory reference analysis. In *ACM Int. Conf. on Supercomputing*, pages 274–284, New York, NY, USA, 2002. ACM. 80

[134] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioglu. On Optimal Parallelization of Arbitrary Loops. *J. of Parallel and Distributed Computing*, 11:130–134, 1991. 78

[135] Optimize with shark: Big payoff, small effort. developer.apple.com/tools/sharkoptimize.html. 71

[136] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Sketching stencils. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2007. 79

[137] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40(5):404–415, 2006. 80

[138] A. Srivastava and A. Eustace. Atom - a system for building customized program analysis tools. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 196–205, 1994. 71

[139] T. L. Veldhuizen and A. Lumsdaine. Guaranteed Optimization: Proving Nullspace Properties of Compilers. In *Int. Symp. on Static Analysis*, volume 2477 of *Lect. Notes in Computer Science*, pages 263–277. Springer-Verlag, 2002. 10

[140] F. P. T. Zaharia, M. Preda. Interactivity, reactivity and programmability: advanced mpeg-4 multimedia applications. In *IEEE International Conference on Consumer Electronics (ICCE'2006), Las Vegas, NV*, pages 441 – 442, January 2006. 62

[141] TeraOps Embedded Project. `http://teraops-emb.ief.u-psud.fr/index.php`. 80

[142] J. Thatcher and J. Wright. Generalized finite automata with an application to a decision problem. *Mathematical System Theory*, 2:57–82, 1968. 22

[143] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the Limits of Program Parallelism and its Smoothability. In Wen-mei Hwu, editor, *ACM/IEEE Int. Symp. on Microarchitecture*, pages 10–19, Portland, OR, Dec. 1992. IEEE. 10

[144] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P.-A. Wacrenier. An Efficient OpenMP Runtime System for Hierarchical Architectures. In *International Workshop on OpenMP (IWOMP)*, pages 148–159, Beijing,China, 6 2007. 80

[145] W. Thies. Streamit: A language for streaming applications. In *Programming Languages and Systems Symposium*, Yale, Connecticut, Aug. 2002. 7

[146] W. Thies, F. Vivien, J. Sheldon, and S. P. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 232–242, 2001. 38

[147] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Chicago, 2005. 31, 78

[148] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 277–288, Chicago, IL, June 2005. 78

[149] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. In *J. of Parallel and Distributed Computing*, pages 139–151, February 1995. 80

[150] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. *J. of Instruction-Level Parallelism*, 7, Jan. 2005. 10, 11, 15, 49

[151] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *ACM Int. Conf. on Supercomputing*, pages 335–344, Cairns, Queensland, Australia, 2006. ACM. 50

[152] N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 292–304, Sept. 2007. 80

[153] VTune Performance Analyzer. www.intel.com/software/products/vtune. Intel. 71, 77

[154] K.-Y. Wang. Precise Compile-Time Performance Prediction for Superscalar-Based Computers. *ACM SIGPLAN Notices*, 29(6):73–84, June 1994. 10

[155] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984. 28

[156] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *ACM Int. Conf. on Supercomputing*, 1998. 19, 66, 78

[157] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`)". 15, 33, 49

[158] 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. 10

[159] L. M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, July 1992. 27, 28

[160] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining Loop Transformations Considering Caches and Scheduling. *Int. J. of Parallel Programming*, 26(4):479–503, 1998. 11

[161] M. Wolfe. Iteration space tiling for memory hierarchies. In *SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics. 37

[162] B. Wylie, B. Mohr, and F. Wolf. Holistic hardware counter performance analysis of parallel programs. In *Parallel Computing*, Malaga, Spain, September 2005. 71, 81

[163] J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: A language and compiler for dsp algorithms. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 298–308, 2001. 30, 78, 79

[164] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzarán, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 63–76, June 2003. 49

[165] K. Yotov, X. Li, G. Ren, M. Garzarán, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLASs? *Proc. of the IEEE*, 93(2):358–386, February 2005. Special issue on "Program Generation, Optimization, and Adaptation". 33

[166] K. Yotov, K. Pingali, and P. Stodghill. X-Ray: A Tool for Automatic Measurement of Architectural Parameters. In *Intl. Conf. on Quantitative Evaluation of SysTems*, 2005. 41, 77

[167] W. Zhang, B. Calder, and D. Tullsen. An Event-Driven Multithreaded Dynamic Optimization Framework. In *International Conference on Parallel Architectures and Compilation Techniques.(PACT)*, September 2005. 73

[168] M. Zhao, B. R. Childers, and M. L. Soffa. A Model-Based Framework: An Approach for Profit-driven Optimization. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, San Jose, California, Mar. 2005. 10