Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Machine Learning of Compiler Heuristics for Parallel Architectures

Doctoral Dissertation of:
Michele Tartara

Advisor:
Prof. Stefano Crespi Reghizzi

Tutor:
Prof. Andrea Bonarini

Chair of the Doctoral Program:
Prof. Carlo Fiorini

2012 - XXV edition

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32 I 20133 — Milano

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Machine Learning of Compiler Heuristics for Parallel Architectures

Doctoral Dissertation of:
Michele Tartara

Advisor:
Prof. Stefano Crespi Reghizzi

Tutor:
Prof. Andrea Bonarini

Chair of the Doctoral Program:
Prof. Carlo Fiorini

2012 - XXV edition

To friends and friendship.

In memory of Leonardo.

Acknowledgements

Three years as a Ph.D. student can hardly be contained in a thesis: it can just show the research you did, not all the people you have met.

I am deeply convinced that you should learn something from every person you meet, and my Ph.D. studies at Politecnico di Milano greatly enhanced my possibility to do so, by allowing me to travel a lot, meeting many great people from all around the World. Listing and thanking all of them in this small space is not possible, so I will name just a few. To everybody else still goes my silent and heartfelt "thank you!".

Until a few years ago, I never thought I was going to work on languages and compilers. Professor Stefano Crespi Reghizzi, my Advisor for the Master thesis and during these three wonderful years, made me realize how interesting these fields can be and let me join his research group to actively contribute to their advancement. I want to thank him because of this, and because he always allowed me a great degree of freedom in choosing my research topic and, at the same time, he never left me without his precious guidance and suggestions, especially when the good results seemed farther away and harder to reach.

I would also like to thank Grigori Fursin, of INRIA, who reviewed the first draft of this thesis and gave me many interesting and extremely useful suggestions on how to refine it towards this final version, and all the anonymous reviewers of the papers that are part of it, for giving me precious hints along all the path of this research.

My praise goes to the "Fondazione Fratelli Agostino Enrico Rocca" for funding the "Progetto Roberto Rocca Doctoral Fellowship" that allowed me to spend 7 months at the Massachusetts Institute of Technology. I am very grateful to my MIT host, Professor Saman Amarasinghe, for having me in the COMMIT research group and for all his priceless suggestions and guidance, to Jason Ansel for starting the PetaBricks project, and to Professor Una-May O'Reilly for the insightful discussions and recommendation.

A huge thank you to all the awesome "Rocca Fellows", and especially Andrea Ratti, for sharing with me a big part of this wonderful experience.

The time I spent in Boston has had a huge impact on my life, and I will always remember it dearly, mainly because of all the new friends I met there. First of all, Stefano ”the social chair” Maffina. Through him I got to know most of the people that made my stay there such a great experience: Laura, Tasmina, Noelle, Mariana, Kerry, Elijah, Bekka, Kyle, Juan, Tatiana, Erik, Siyang, Harrison, to name just a few of them.

Also, my great housemates have to be remembered (in order of appearance): Guido, Björn, Tom, Andrea, Julia, Daniele. Living with you all has been a ton of fun and a great opportunity of getting to know really well people from abroad.

Going back to Italy, I have to thank all people I first met in Office 127 at Politecnico, for all the inspiring and fun time spent together.

In particular, in Simone Campanoni I found a great colleague, counsellor and friend, even more so since my stay in Boston. His neverending enthusiasm (some would term it “optimism”) has always been greatly inspirational to me, and thanks to him I started to delve into the fascinating worlds of compiler backends and ARM processors.

Andrea Di Biagio, Giampaolo Agosta and Gerardo Pelosi, enlightened my path through their experience and knowledge, with lots of priceless hints.

Spending three years next to Ettore Speziale, sharing all the steps of our Ph.D. studies has been a great experience, that taught me a different approach to programming, doing research, and... writing song lyrics!

To Michele Scandale, the newcomer of the office, thanks for joining the group with his enthusiasm, his technical skills and his witty jokes.

To Alessandro “O Captain! My Captain!” Barenghi goes a very special “thank you”, for being a great friend, a neverending source of interesting facts and stories, and, of course, for the (experimentally proven) entangled neuron. Also, thanks for introducing me to the world of international computer security competitions and thanks to all the members of the “Tower of Hanoi” team for all the inspiring and challenging days (and nights) spent together.

Thanks to all the friends that shared with me part of these three years: Vito, Nick, Flavio, Silvia, life at DEI without you would have been much less enjoyable!

And friends are important outside the university, too. To all the members of the ”Sgruppo” (in no particular order, Luca, Letizia, Michele, Leonardo, Simona, Marco, Eleonora, Elisa, Matteo, Thomas, Alice, Silvia, Antimo, Silvia) thanks for always being next to me, no matter the physical distance.

Finally, but first and foremost, thanks to my wonderful Parents and to my family for their neverending support and guidance, and for helping me achieving my dreams.

Sommario

Le architetture dei calcolatori sono in continua evoluzione: guadagnano nuove funzionalità e diventano più veloci, e complesse ogni volta che ne viene rilasciata una nuova.

Al fine di sfruttarle pienamente, è necessario che anche i compilatori vengano aggiornati, per permettere ai programmatori di avere pieno accesso a tutta la potenza di calcolo fornita da tali architetture moderne.

Sfortunatamente, mentre la legge di Moore predice che il numero di transistor nei processori raddoppi circa ogni due anni, la legge di Proebsting ci dice che il livello di ottimizzazione fornito dai compilatori è previsto raddoppiare ogni diciotto anni. Tali numeri ci danno una chiara idea di quanto sia complicato per i compilatori tenere il passo delle innovazioni introdotte dall'hardware.

Il problema principale è che molte ottimizzazioni di compilazione possono fornire sia un miglioramento sia un peggioramento delle prestazioni, a seconda del codice a cui sono applicate, e a seconda di quali altre trasformazioni vengono applicate prima e dopo quella considerata. Decidere se e quando applicare un algoritmo di ottimizzazione è un compito estremamente complesso, e la complessità delle architetture rende impossibile l'uso di modelli esatti per predire il risultato, perciò i compilatori utilizzano euristiche per prendere tali decisioni.

Il processo di scrittura delle euristiche è, tradizionalmente, basato soprattutto sull'esperienza personale del compilatorista e include un lungo processo di evoluzione delle stesse per prove ed errori. Molti lavori si sono occupati di recente di provare a sostituire al compilatorista degli algoritmi automatici per svolgere questo compito. A tale scopo sono state sviluppate la compilazione iterativa e algoritmi di apprendimento automatico applicato alla compilazione: ognuno di questi due approcci presenta pregi e difetti.

Questa tesi si colloca in quest'area di ricerca. Prima di tutto, essa presenta *long-term learning* (apprendimento di lungo termine), un nuovo algoritmo di apprendimento che mira a definire automaticamente delle euristiche di compilazione efficienti. Long-term learning prova a superare i principali ostacoli della compilazione iterativa e degli approcci di apprendimento automatico (i lunghi tempi di compilazione e il bisogno di svolgere una lunga fase di addestramento iniziale) e al tempo stesso ne mantiene i vantaggi. Sfide simili sono già state affrontate da recenti lavori dell'area, ma, unica dell'apprendimento di lungo termine è la capacità di risolverli generando euristiche facilmente comprensibili (nella forma di formule matematiche) e tenendo in considerazione il fatto che i singoli algoritmi di trasformazione del codice non sono indipendenti, quindi le loro euristiche necessitano di essere evolute in modo tale da interagine bene le une con le altre.

Al fine di velocizzare ulteriormente l'esecuzione dell'algoritmo di apprendimento di lungo termine, questa tesi presenta un metodo per parallelizzarlo su più macchine, o per eseguirlo in parallelo su una singola macchina suddividendone le risorse, usando un approaccio basato su MapReduce. Questo approccio non è limitato all'uso su long-term learning, ma è generale e può essere applicato alla maggior parte degli algoritmi di compilazione iterativa.

Infine, vengono presentate due proposte di lavori futuri. Primo, un nuovo metodo leggero di compilazione per programmi altamente dinamici, che permette di suddividere il processo di compilazione tra compile-time e runtime, mantenendo quanto più possibile i calcoli più pesanti a compile-time ed applicando a runtime solo quelle trasformazioni che potrebbero beneficiare della disponibilità di ulteriori informazioni non disponibili in precedenza, uando una tecnica derivata dall'apprendimento di lungo termine per determinare quali ottimizzazioni posticipare a runtime. Secondo, a partire da un analisi della sensibilità delle primitive di sincronizzazione (lock e memorie transazionali) ai guasti hardware, viene proposto un uso dell'apprendimento di lungo termine come base per un nuovo approccio al recupero dai guasti.

Abstract

Computer architectures are continuously evolving: they become faster, more feature rich and more complex every time a new one is released.

In order to fully exploit them, compilers have to be updated as well, to allow the programmers to have full access to all the computational power these modern architectures provide.

Unfortunately, while Moore's law predicts that the number of transistors in processors doubles roughly every two years, Proebsting's Law tells us that the optimization level provided by compilers can be expected to double every eighteen years. Such numbers give us a clear insight about how difficult it is for compilers to keep the pace with the innovations introduced by the hardware.

The main problem is that most compiler optimizations can provide either speedups or slowdowns depending on the code they are applied to and depending on which other transformations are applied before and after the one being considered. Deciding whether and when to apply an optimization algorithm is a daunting task, and the complexity of the architectures makes the usage of exact models to predict the outcome unfeasible, so compilers rely on heuristics to make such decisions.

The process of writing heuristics is, traditionally, mostly based on the personal experience of the compiler writer and involves a time-consuming trial and error process. Much work has been done recently to try and substitute the compiler writer with automated algorithms for performing this task. Iterative compilation and machine learning approaches have been studied, each with its own merits and shortcomings.

This thesis work is rooted in this research area. First of all, it presents long-term learning, a new learning algorithm that aims at automatically determining efficient compilation heuristics. Long-term learning tries to overcome the main issue of most iterative compilation and machine learning approaches (the long compilation times and the need for a time-consuming initial training phase) while still providing their advantages. Similar challenges have been faced already by recent works in the area, but, unique to long-term learning is the ability to solve them while generating human-readable heuristics (in the form of mathematical formulas) and while taking into consideration the fact that the single code transformation algorithms are not independent, therefore their heuristics need to be evolved in such a way to interact well with one another.

In order to further speedup the execution of the long-term learning algorithm, this thesis presents a method to parallelize it across multiple machines or to execute it in parallel on a single machine by splitting up its resources, using an approach based upon MapReduce. This approach is not limited to long-term learning, but it is general and it can be applied to most iterative compilation algorithms.

Finally, two proposals are presented as future work. First, a new lightweight compilation method for highly dynamic parallel programs, allowing one to divide the compilation process between compile-time and runtime, keeping as much as possible of the heavyweight computations at compile-time, and applying at runtime only those transformations that could benefit from information not available at compile-time, using a technique derived from long-term learning to choose which optimizations to postpone at runtime. Second, starting from an analysis of the sensitivity to hardware faults of synchronization primitives, namely locks and transactional memories, a use of long-term learning as the basis for a novel approach to fault recovery is presented.

Contents

List of Figures

List of Tables

1 Introduction

The complexity of modern computer architectures is constantly increasing every time a new one is released. Because of this, compiler writers cannot know in advance the exact effect of applying a given code transformation, since too many characteristics (size of the cache memories, pipeline length, degree of parallelism, *etc.*) actually influence the result. This is especially when multiple interacting optimizations are considered.

Building an exact model of such architectures that can be used by compiler to decide what optimizations to apply to a program, in which order to apply them, or what values to assign to numerical parameters required by the compilation process is computationally infeasible. Therefore, all compilers use heuristics to make such decisions, based on program-dependent values available at compile-time, called *features*. Possible features are the average number of instructions in a basic block, the nesting level of a loop, the average number of successors for each instruction, *etc.*

Usually, compiler experts write the heuristics manually. This is a time-consuming task, and it is likely to yield suboptimal results since it is entirely based on the skills of one or few experts. Deep knowledge of the functioning of the architecture is required, including being aware of how the memory hierarchy can provide data in an optimal fashion, and how the functional units can be used together with the register to provide the correct operands at the correct time, preventing avoidable lag. Also, many parameters such as software pipelining strategies, instruction scheduling, blocking factors and loop unrolling depths have to be decided. Moreover, to obtain the best results on all architectures, the process of defining the heuristics should be repeated for every time a new version of the compiler is released, or a new target architecture becomes available, since different platforms require different optimizations and therefore different heuristics. It might take several releases of the compiler to exploit all the features of a new architecture, and, given the current rate of evolution of the hardware, by the time the compiler is ready, a new architecture is likely to be available.

During my Ph.D. I had the occasion of working on the internals of many different compilers: GCC, PetaBricks [5], ILDJIT [18] and, to a lesser extent, LLVM [78]. I started to consider the importance of simplifying and speeding up the task of adapting a compiler to multiple platforms and to different systems while having my first experience working on compilers. In particular, while I was adding ARM support to the Libjit code generation library [117] and porting the ILDJIT compiler to work on the NHK-15 ARM platform [118] by STMicroelectronics.

Fully supporting all the characteristics of a given platform can take a long time, and even when all the optimization algorithms are ready, more time has to be spent developing the decision making algorithms responsible for activating the optimizations, determining their application order and selecting the numeric parameters (if any) needed by the optimization algorithms. Furthermore, the focus of the optimization process keeps changing [23]: in the 1980s the speed of the code was the main objective. In the late 1990s, with the embedded systems becoming more widespread, the size of the compiled code became more important. Later, embedded systems also determined interest in optimizing programs to reduce their power consumption.

In order to face these issues, researchers proposed various techniques over time. Iterative compilation is the simplest one. It is a technique used to produce better, more optimized, programs by compiling hundreds of versions of each of them using different optimization settings, and then keeping the best one as the result of the compilation process.

Usually, the objective function used to determine which program is the best one is based on running the programs while computing their execution times in order to find the fastest one. Still, other objective functions are possible, like minimizing the size of the generated code or minimizing the compilation time. Finally, it is also possible to have a multi-objective function, aiming at minimizing more than one of these dimensions.

The various versions of the program being compiled are created by applying a different compiler configuration, one for each version, to the source code being compiled.

Depending on the specific iterative compilation algorithm considered, a configuration can comprise different things. It can determine the set of optimizations to be used, it can describe the order in which the optimization algorithms are applied, or it can define some compilation parameters (*e.g.* loop unrolling factor, size of the tiles for a matrix tiling algorithm, limits for function inlining, *etc*). The choice of which configurations to try is determined by random selection or by simple space exploration algorithms.

The main issue with iterative compilation is its slowness. Looking for the best version of a program by compiling and testing multiple candidate versions requires a lot of time, and the overhead introduced by is too high for most applications. Nevertheless, the technique is useful in some specific areas, such as when dealing with embedded systems. Here the available resources are scarce: the battery charge is limited, the memory is not abundant and the processors are not too powerful, so optimizing the programs is of paramount importance. Furthermore, on embedded devices a program is likely to be deployed on many identical systems. Therefore, the wide impact of each performance improvement obtained, however small, makes the extra compilation time worth spending.

In order to reduce the time required by iterative compilation, several researchers have introduced machine learning techniques that aim at restricting the search space of possibly profitable versions, in order to reduce the amount of iterations needed. Machine learning techniques speed up the search at compile time, by building a model of the system where the compilers are run: this model is then used to predict the more profitable optimizations to be applied, without the need of actually running the executable files they produce. Unfortunately, machine learning techniques require the model to be trained when the compiler is installed on a new machine (that is, *offline* or at *deploy time*), before being used. This is an extremely time-consuming operation, that can last for weeks [46], and this limits the applicability of machine-learning approaches.

Reducing the size of the space to be explored is the most important way of improving iterative compilation techniques, but it is not the only one. Various other methods have been proposed to speed up the process. Leather, in his PhD thesis [80], suggests to use a statistical approach based on confidence to determine when to stop sampling new timing runs, in order to remove noise without performing useless iterations. Fursin et al. [42] exploit the structure of particular programs that present relatively stable phases, to compare multiple versions of the code within a single run.

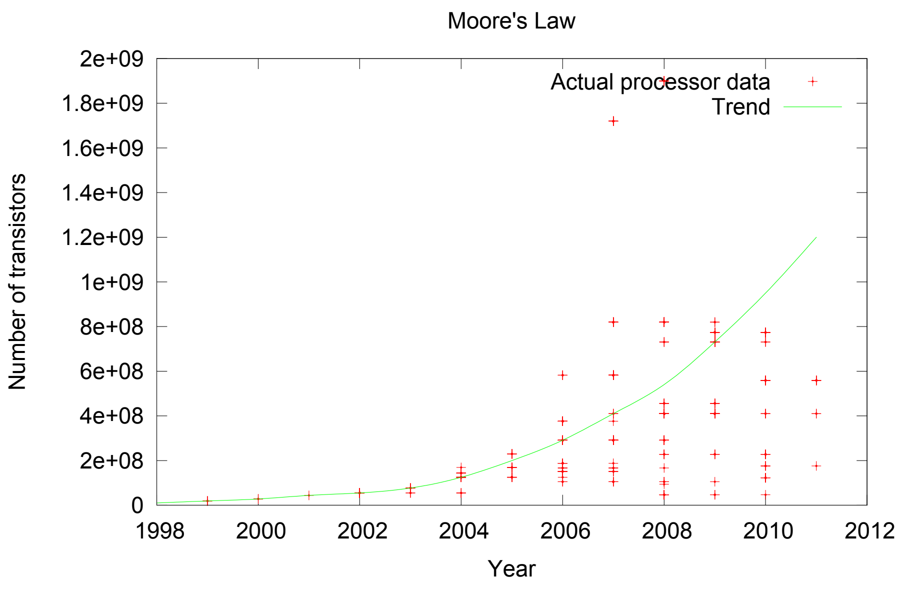


Figure 1.1: Moore's Law, stating that the number of transistors in a processor doubles every two years. The plotted data are taken from Intel website and are about Intel processors only.

All these techniques are effective on their own, but they still require a large number of programs to be executed to compare their performance.

Most of the work presented in this thesis will deal with developing new techniques for performing iterative compilation based on machine learning, automatically generating human-readable heuristics while using a reduced number of candidates.

Furthermore, trends in computer science show an ongoing shift of paradigm, from *sequential* to *parallel*, because of the inability to further increase clock rates due to technological and thermal issues [16], and for exploiting the improved transistor density guaranteed by Moore's law.

Moore's law, formulated by Gordon E. Moore in 1965, predicted that the number of transistors of a CPU could be expected to double each year [91]. Later, he refined his prediction, saying that the doubling of the number of transistors could be expected every couple of years [92]. The data in Figure 1.1 show that this still holds nowadays.

With multi-/many-core processors, the offered parallelism evolved from an *implicit* form (*e.g.:* Out-of-Order execution [124]) to an *explicit* form, where processing elements are directly controlled by programmers.

From an architectural perspective, this allows simplified processor design (by removing power- and area-hungry components, like branch predictors [131]) and freeing up resources to increase the number of processing elements. This approach has been exploited in GPGPU designs [98], where there is a wide number of processing element very similar to an ALU. This leads to an increased importance of the compiler: simpler processors are less able to reschedule instructions, therefore the compiler has to be smart at defining their execution order.

The push toward explicit parallelism influences programming models, compilers and runtimes. Research has shown that no known technique or methodology can deal with all the available parallel architectures and challenges [9]. All the explicit parallel programming models push programmers to define elementary units of work, called either *task*s [39], or *parallel iteration*s [99] or *work item*s [65], and let the compiler and/or the runtime schedule them according to their mutual dependencies.

In this context, compilers have been used in a classical way: they perform translation to machine code, that is later executed. Leveraging on the structure of the input language, some compilers perform aggressive optimizations, such as work-items pre-scheduling [63, 114]. The generated code is always oblivious to the existence of a compiler, and the compiler, even when it is a JIT, does not provide any kind of "service" to the program.

As a future work, I will presents a more dynamic approach to compilation, where the generated code is meant to interact with the compiler during execution, to exploit dynamically available information to optimize the code on-the-fly, without the burden of a full-fledged JIT system. The idea of splitting the compilation process between runtime and compile time has already introduced in [62], but in the approach I present, the compiler works together with a micro-thread scheduler. *Stalls* in micro-threads containing the program code can be used to execute the compiler optimizers, thus *minimizing compiler overhead* at runtime. The decisions about how to prepare the code for compiler interaction will be taken using an approach based on machine learning techniques.

The main motivation for pursuing this approach is code optimization: by running the compiler during code execution, more information is available, enabling more precise optimizations. At the same time, the machine learning approach could continuously improve the quality of the generated code, allowing it to be fitter to be specialized at runtime.

As a side effect, this approach would benefit software deployed in binary-only form, meant to run on many different hardware configurations (*e.g.:* binary packages used by Linux distributions). Allowing the program to customize itself without being compiled from a bytecode form at deploy-time and without the need for a Just-In-Time compiler (that is time- and resource-consuming at runtime) could be useful in a variety of scenarios.

This thesis will present the solutions and algorithms I developed to solve some problems related to machine learning and iterative compilation and the just described lightweight compilation technique that could benefit from such algorithms.

In particular, in Chapter 3 introduces long-term learning, a new learning algorithm for compilers that smooths out the biggest slowness-related issues of both iterative and machine-learning-based compilation. In particular, it drastically reduces the long compilation times typical of iterative compilation drawbacks, and, at the same time, completely removes the need for an initial training phase. The chapter shows that long-term learning is able to improve the performance of a compiler over time, thus allowing us to remove the initial training phase. Finally, it shows that different compilers can benefit from the use of long-term learning, by presenting two implementation and experimental results gathered on multiple hardware configurations. The implementations are on top of the PetaBricks research compiler and the GCC production-quality compiler.

Chapter 4 presents a novel approach based on MapReduce that is able to improve the performance of iterative compilation techniques by parallelizing the testing of the candidate programs on multiple machines taken from an homogeneous cluster. The chapter also presents a method that allows one to exploit the parallelism available in modern machines (both UMA and NUMA ones) by applying the proposed approach to parallelize the iterative compilation of sequential or moderately parallel programs. Finally, the chapter includes experimental evidence of the efficacy of the approach, obtained through a large and diversified experimental campaign performed on various hardware configurations. This parallelization approach was specifially developed for PetaBricks, and the experimental data are gathered using such a compiler. Though, the approach is completely general and can be applied to every iterative compiler.

Chapter 5 shows that, even using iterative compilation and machine learning, not every optimization can be applied at compile time. Usually this happens when an optimization algorithm depends on some constant that is only known at runtime, or when the optimization is determined by the input data of the program. The chapter therefore describes a future work possibility where a lightweight compilation approach performs as much as possible of the optimization process at compile-time, and prepares the code to be further optimized by the runtime library using a micro-threaded approach. The selection of which optimization algorithms should actually prepare the code to be modified at runtime requires the use of heuristics that can be learned by the long-term learning algorithm. In the same chapter, another possible future work is described. Given an analysis of the sensitivity to hardware faults of synchronization primitives used to write parallel programs, a method to exploit machine learning techniques similar to those presented in Chapter 3 to correctly exclude the faulted parts of the system and still manage to exploit the working ones.

2 Related Work

Most compute-intensive or memory-intensive programs can benefit in speed from being tuned to work on a specific computer architecture. Unfortunately, if the tuning process is performed by hand, it usually takes a very long time to optimize a single program. Nevertheless, there are certain heavy-duty scientific computational libraries and programs that are used for computationally intensive and time-consuming tasks and it is of paramount importance for them to be as optimized as possible. Because of this, the first works in the field of iterative optimization were born to optimize them. Such works were not general: they did not directly involve the compiler and were specifically targeted at the single library, by means of multiple implementations of certain algorithms, selection of compile-time constant parameters and hand-crafted makefiles to drive the compilation process.

Only at a later time researchers began to modify compilers, to allow this kind of optimization to be performed on a wider variety of programs.

The rest of this chapter is structured as follows. Section 2.1 presents the early works using iterative techniques to optimize programs and libraries: not all of these early works directly involve a compiler. Section 2.2 describes iterative compilation and Section 2.3 presents the evolution of iterative compilation with the introduction of machine learning techniques. As shown in Section 2.4 and 2.5 these are not the only approaches to adapt an application to its target architecture. Just-In-Time compilation and autotuning also exist and are active research areas. Finally, Section 2.6 presents certain techniques that are not directly related to compilation but have been used to develop the work presented in the other chapters of this thesis.

2.1 Execution-driven approaches

The early works related to iterative compilation did not directly involve compilers, but used iterative techniques to optimize programs and libraries. These techniques were based upon building, executing and evaluating multiple implementations of the algorithms, and can therefore be defined as *execution-driven approaches*.

An example can be found in [129], where the authors describe the development of ATLAS (Automatically Tuned Linear Algebra Software), a library API-compatible with the BLAS library (Basic Linear Algebra Subroutines) [32, 31], using autotuning techniques to improve its performance. In particular they present their work on the general matrix multiply function, *DGEMM*. The ATLAS approach consists in isolating the machine-specific sections of the code to a restricted set of routines. The code of such routines is created by a code generator, that varies the parameters according to some timing runs. The search for the optimal parameters can be sped up by the user, providing information about the underlying hardware architecture, such as the cache size or the blocking factors to try. The rest of the code is architecture independent and does not influence the performance. The basic idea of the ATLAS library is to find the fastest and biggest on-chip multiply (that is, an algorithm able to perform the matrix multiplication without spilling to the system memory) and then using it as a building block for obtaining a general and efficient matrix multiplication implementation. This is done by fitting the algorithm into the L1 cache, reordering the floating point instructions to hide latencies, reducing the loop overhead through loop unrolling, and exposing the parallelism of the hardware.

Another example of application-specific use of iterative tuning techniques is OSKI (Optimized Sparse Kernel Interface) [128]. It is "a collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices". Its tuning process determines the best data structure to use on a given architecture to represent the sparse matrices. It features self-profiling techniques that are used to allow a program to gather information about the matrix operations that are performed, and to use such information to determine how aggressively to tune and to guess whether tuning will be profitable. OSKI is meant to tune the program not only to fit the hardware, but also to fit the specific input data. Therefore, its tuning process happens at runtime, and it accepts hints from the user about the data and the target architecture to reduce the runtime overhead.

A more general approach was presented by Diniz and Rinard [30], who worked at the compiler level. They introduced *dynamic feedback*, a technique enabling the compiler to generate a single executable containing multiple versions of the same source code, each compiled using a different optimization policy. During the execution, at fixed time intervals, the program switches between a sampling phase and a production phase. While in the sampling phase, all the various versions are executed, in turn, and their performance is measured. When the system is switched back to the production phase, the version that proved to be the best is used. This approach makes the programs able to adapt to dynamically changing environments, where either the available computational resources vary considerably during the execution, or where the data sets to be elaborated during a single execution of the program heavily differ from one another. The main drawback of this approach is in the fact that the executables tend to be quite bloated, since they need to contain multiple versions of the binary code. Furthermore, there is no feedback from the actual selection performed at runtime back to the compiler. Therefore, the compiler cannot exploit this information to improve the code generated for the next programs it compiles: it can only guess which optimizations to use during the code generation according to some pre-defined heuristic.

2.2 Iterative Compilation

To the best of my knowledge, iterative compilation was first introduced as the main innovation of the OCEANS project [13], and is first thoroughly described in the works by Kisuki et Al. [67] and Bodin et Al. [15].

Iterative compilation consists in generating many different binaries from the same source code, using different compiler configurations that enable different subsets of the available optimization algorithms, or that select numeric parameters for optimizations. All the configurations are then tested on the same dataset, and then the one that proved to perform better is selected as the result of the compilation process. The search space of the possible compiler configurations is too big to be explored entirely, therefore only a subset of some hundreds of configurations is chosen.

The two papers use different amounts of iterations in different experiments, but [67] argues that for most programs 200 iterations are usually enough to reach the maximum performance. The choice of which configurations to test is not completely random, but uses an extremely simple algorithm. The rationale for using iterative compilation was the observation that embedded systems were increasingly using general-purpose processors, therefore their software was being written by a compiler instead of being hand-crafted. Still, the resources of embedded systems were extremely scarse, and the need to obtain optimized code was strong. The same considerations still hold today, but the speed of current systems makes the iterative approach and its later descendants useful also for high performance computing and, in some cases, for software in general.

Both in [67] and [15] iterative compilation is used just to select two parameters: the unrolling factor for the loop unrolling optimization and the tiling size for the loop tiling algorithm. These two parameters are critical because the two optimizations have competing objectives, and modifying one of them has a strong impact on the performance of the other algorithm: finding the right balance between the two is, therefore, difficult.

Also, [67] hints about the need for multi-objective optimization (to produce executables that are both fast and small, as later done by [58]) and about the improvements that could be obtained by using a model to select the most promising optimizations (as later implemented by the machine learning-based compilation techniques described in Section 2.3).

Nisbet [96], almost at the same time, introduced the use of evolutionary algorithms (presented in detail in Section 2.6.1) to guide the selection of the configurations to generate. In particular, in his GAPS system, he used a genetic algorithm to search the solution space of legal, ordered sets of transformations to be applied to the source code. The implemented approach is not a 100% pure genetic algorithm approach, though, because it also allows the user to provide specific knowledge about the program being compiled and the architecture it will be run on. Also, the approach is not particularly efficient: the full learning process is run every time a new program is compiled, and it needs some thousands of iterations to find a good solution. Better techniques to determine good candidate configurations to try will be introduced by later works, and are described in Section 2.3.

After [67] and [15] showed the feasibility of iterative compilation, and [69] further expanded that work, [68] extended their research by considering another transformation, array padding, next to loop tiling and loop unrolling. Furthermore, it details a general, although really simple, algorithm for learning a theoretically unlimited number n of parameters through iterative compilation. The search space is divided into a n-dimensional coarse grid. One candidate is tested for each point of the grid. In the area surrounding the best candidates the grid is refined with a new, more fine-grained n-dimensional grid, and the procedure is repeated recursively. The authors test the candidate programs using both a single data size and multiple data sizes for the input, and show that the use of a single data size is sufficient to obtain good performance on every data size of the actual input. Therefore, they suggest using small input sizes for the testing, to reduce compilation times, without impairing the performance of big input datasets during the actual usage of the program. They claim to obtain an average improvement of 35% over existing techniques by using iterative compilation over 400 candidate configurations.

[66] is the first work dealing with the choice of compiler switches to enable and disable the optimization algorithms through an iterative compilation approach. This kind of decision requires the exploration of a huge search space, so it is particularly important to be able to prune it in an efficient way. The paper uses a method called *fractional factorial design*, that systematically designs a series of experiments that determine the switches with the best chance of being optimal. It also aim at precisely quantifying the interactions among the switches, to better determine which combinations to actually explore.

The order in which the compiler optimization passes (or phases) are applied is important and changing it can heavily influence the final result of the compilation. Kulkarni et al. [77] tackle this problem by using a genetic algorithm. The big size of the search space that needs to be analyzed leads to a compilation process that could last for hours or even days to find the best resulting binary. So, the authors focus on trying to reduce the number of candidate programs to test. They consider the program functions as the compilation units and apply various techniques to avoid testing the same code multiple times. Among these, the following ones are worth mentioning. First of all, since the genetic algorithm can happen to generate the same optimization sequence more than once, a hash table of attempted sequences is maintained, along with the performance results of each sequence, so that there is no need to test the same sequence twice. Analogously, it is not infrequent for different sequences to generate the same binary, usually because some of the passes cannot be applied to the given program and, therefore, their position in the sequence has no influence on the final result. So, a table is kept, with the hashes of the generated binaries and the associated performance results. Identical binaries will not be tested again. In a similar way, they deal with functions consisting of equivalent code, that is, code made up of the same instruction, but with a different assignment of registers.

The main issue of iterative compilation is the long time required to test all the different versions of the binary produced from the same source code. The method proposed by [42] aims to face this issue by using a method that vaguely resembles the one introduced by [30]. The authors argue that many programs exhibit a behavior that can be modelled with phases of stable performance. They implemented a low-overhead performance stability/phase detection scheme and they generate executables containing multiple versions of the code. They take advantage of phase intervals with the same performance to evaluate a different optimization configuration at each interval. This way, each executable can compare multiple candidates. The intervals are not of fixed length, but are determined by the detection of stable phases. This method is used as a building block for a more traditional iterative compilation algorithm: the results of each set of comparisons is stored in an archive and is later used to pick the overall best result. This is particularly useful for applying iterative compilation to programs that require very long running times, because it allows to greatly reduce the number of executions to perform. The authors also suggest that their approach could be used for designing self-tuning applications, able to adapt themselves to the data being elaborated, by switching different versions of the code. The biggest shortcoming of the presented approach is the increased code size, so they suggest using a big number of alternative versions when looking for the best one as a step of an iterative compilation approach, and using only a few of them if implementing an autotuning program. This research is further expanded in [44] and [43].

The research on iterative compilation focused mainly upon reducing the execution time of the programs. Though, in some cases, execution time is less important than code size, especially for those embedded systems where the program has to be burnt into a ROM: having a smaller program could enable the use of a smaller and cheaper ROM. Cooper et al. [22] use a genetic algorithm to look for an optimization sequence able to reduce the size of the compiled program.

Later, in [23] and in [3], they characterize the space that an adaptive compiler must search to perform its duty, and show that *biased random sampling* is able to find good solutions quickly. With "biased random sampling" they describe all the algorithms such as genetic algorithms and other iterative compilation algorithms where the selection of new candidates is random but biased by the knowledge of which candidates already performed well in the past.

Most iterative compilation approaches are used to optimize a single objective function, usually the speed of the compiled executable. Nevertheless, multiple dimensions are actually an interesting target for optimizations, such as the size of the final executable, the compilation times, the energy consumed during the execution of the program, *etc.* Hoste and Eeckout [58] introduced COLE (Compiler Optimization Level Exploration), a framework for automatically finding Pareto optimal[1] optimization levels through multi-objective evolutionary searching. To the best of my knowledge, this is the first work dealing with multi-objective optimization of programs through iterative compilation. The paper demonstrates that the automatic construction of Pareto optimal optimization levels is feasible in practice, and performs an analysis of such solutions to determine the importance of the various compiler optimizations, finding that only some of them are actually present in the Pareto optimal optimization levels, and only a few appear in all the levels.

The problem of finding good optimizations with iterative algorithms takes a really long time. Along with the presented approaches, aiming at reducing the number of iterations, other solutions have been devised, based on the machine learning techniques presented in the next section.

2.3 Machine learning algorithms

Several researchers have introduced machine learning techniques that aim at restricting the search space of possibly profitable optimizations to apply to obtain good code.

The basic idea is to reduce the number of iterations and the amount of time needed by iterative compilation. This is done by learning a model of the target system. After the training, the model is used to predict what optimizations to use. Depending on the specific approach, either a set of likely good optimization configurations are compared, or just the one supposed to be the best one is directly used to generate the final executable program.

Machine learning algorithms can be divided into two main categories: offline and online, and they will be presented in the next two sections.

2.3.1 Offline machine learning

Offline machine learning is the most widespread and most studied kind of machine learning approach. It is divided into two distinct phases. The first one is called *training phase*. It has to be executed when the system is being setup. It is needed to allow the system to adapt to the specific environment it will have to work into by learning the characteristics of

[1]a Pareto optimal solution is a solution that can not be beaten by another solution along all objective functions simultaneously

the target machine. The second one is the *deploy phase* (or deploy time) and it allows the compiler to use the learned model to predict the best optimization for a given program.

Various algorithms have been presented for applying machine learning techniques to compilers, and they will be briefly described in this section.

One of the first works dealing with machine learning techniques for iterative compilation is [90] by Monsifrot et al. The authors use static program features to make decisions about the loop unrolling optimization to be applied to fortran programs. An initial training phase is used to learn the model, using a set of loops taken from various programs. The model is a decision tree where each node checks the value of a feature of the program. When a new program has to be compiled, the decision tree is used as a classification process to decide how and if to apply loop unrolling.

Being an early work, the authors also perform a study to verify whether it is actually possible to try and learn a heuristic for a specific architecture. They apply their approach on two different architectures, and obtain good results on both of them. Then, they use the decision tree learned on the first architecture to make decisions for the second one, and vice-versa. In both cases, the obtained speedup drops by more than 90%, showing that the heuristics effectively learn how to make decisions for the specific architecture they are trained on.

The first more complete work about machine learning techniques applied to compilation is [2], by Agakov et Al. They use program features to correlate new programs with previous knowledge. The features they use are characteristics of the programs that can be computed statically. Their analyzer extracts 33 features for each program. The authors observe that the selection of the right features is of paramount importance for the success of this approach, to capture all the possible information about the program. At the same time, using too many features makes it difficult and computationally expensive to use them to learn a model. Therefore, they use Principal Component Analysis (described in Section 2.6.3) to reduce the number of features to only 5, preserving most of the useful information. These five features are used to train two models (for comparison): an independent identically distributed (IID) model, and a Markov model. The first one considers the code transformations to be independent, the second one takes into account the fact that they are mutually interacting. They use an iterative compilation approach based on a genetic algorithm to determine which optimizations to enable and the number of loop unrollings to perform. The machine learned models are used together with the features extracted from the program to be compiled to generate the initial population of the genetic algorithm, thus biasing the random search to a certain area of the search space. The paper presents a three way comparison between the two models and a completely random selection of the initial population. The random search is the worst one. The IID model works well for large search spaces, whereas the Markov model works well on smaller ones. Therefore, the authors suggest an approach using an IID model at the beginning, switching to the Markov model at a later time, when the search space has already been restricted. Furthermore, the IID model can be considered an online probabilistic

Stephenson et al., in [113], observe that a single cost function often dictates the efficacy of a heuristic. Therefore, instead of training a model based on neural networks or classification, they propose *Meta Optimization*, a method by which a machine learning algorithm automatically searches the priority function solution space to directly learn the cost function itself. The cost function is a mathematical formula using the features of the program as variables. To the best of my knowledge, they are the first proposing such an approach. The cost function is learned by using a genetic programming algorithm (as described in Section 2.6.1) that represents the expression as a tree in order to manipulate it. To keep the expressions readable and to prevent it to grow indefinitely, a technique they call *parsimony* is used, that favors short expressions over long ones. The authors show the efficacy of Meta Optimization by providing results over three optimization: hyperblock formation, register allocation and data prefetching. Furthermore, they show that the technique can be applied in two different flavors: either to evolve general purpose heuristics to be later used on any new program, by training the formula over a set of benchmarks, or by creating application specific heuristics, obtained by training them over a single program, that is the one that has to be compiled. This second approach can be seen as a kind of feedback directed iterative compilation.

Later, Stephenson et al. tried a different machine learning approach, based on classification, in [112]. They experimented with two classifiers: a nearest neighbor classifier, and a support vector machine. Both of them are multi-class classifiers: this means they are able to distinguish the input elements in multiple classes, and not just in two. This is fundamental, because the presented experiment deals with learning the unrolling factor for loops, in a scale going up to 8 unrolls. The approach they follow is a kind of supervised learning. An initial training phase is needed (they say that it can be weeks long). For each program, a set of features describing it is extracted, and they are correlated with the best unrolling factor obtainable for that loop, determined by exhaustive analysis of the solutions search space. When a new program has to be compiled, the trained classifier is used to determine the right class (that is, the right unroll factor).

The paper is also interesting because it shows that a correct feature selection is of paramount importance to obtain good results from the machine learning process: having many features can provide much information, but having too many features can make the learning process harder and slower. Therefore, two methods are shown that allow us to find out the best features to use: computing mutual information score[2] and using greedy feature selection[3].

The authors of [101] present another method to predict good optimization sequences: tournament predictor. This predictor uses performance counters to characterize dynamic features of the program and to predict the speedup difference of two given optimization sequences. Given a set of speedup sequences, the best one is found by pairwise comparisons between the currently known best one and all the other ones. The authors compare their predictor with two other state of the art models: a sequence predictor, that is able to determine the probability of each single optimization to be beneficial, and a speedup predictor, able to predict the expected speedup of a given optimization sequence for a program. In their experimental results, the tournament predictor turns out to be the one producing the fastest executable, with the speedup predictor as a close second and the sequence predictor a distant third.

Most of the machine learning-based approaches to compilation use statically computable information about the program being compiled as the features the learning is based upon. [21], by Cavazos et al., argues that static features characterize local code constructs, but provide a poor global characterization once aggregated over many such code sections, and lack the ability to describe the behavior of large control-flow intensive programs. Therefore, they implemented a system where the features are 60 performance counters. The model they create is able to predict what optimizations to use for a never seen before program by using as input features the values of the performance counters of the program itself before being optimized. This approach requires quite an initial training phase to be executed on the target architecture, to build

[2]Using the definition from [112]: "The mutual information score measures the reduction in uncertainty in one variable given information about another variable". Unfortunately, the mutual information score does not tell us anythng about how features interact with each other.

[3]Greedy feature selection identifies features that perform well for a given classifier and a given training dataset by choosing the single feature that best discriminates the dataset, then choosing the second best features that together with the first one best discriminates the dataset, and so on.

the model.

One of the shortcomings of machine learning techniques is that the training phase can be fairly long, and it need to be performed on each new platform the compiler has to be used on.

Thomson et al. [121] try to reduce the training time of the model by focusing it on the programs that best characterize the search space of the optimization sets. They first gather the static features of all the programs in the training set, then apply unsupervised clustering in the program feature space. This way, a classification of the programs according to the similarity of their features is obtained. The most typical program of each cluster is chosen, obtaining a subset of programs that is representative of the whole search space but much smaller. Each of the programs of the reduced training set is then compiled and tested with 4000 different optimizations sets, using a supervised learning technique to learn the model. The model is then used to immediately provide the supposedly best configuration, one-shot, without using iterative compilation. This approach is shown to be 7 times faster than others during the training phase, and is able to provide results close to the maximal during the deploy phase.

Dubach et al., in [33], try to remove the need for the initial training phase over a benchmark suite while retaining the possibility to use a model to find the best candidates to test in an iterative compilation approach. The initial, really long, training phase, is substituted by a short one, specific for each program being compiled. Every time a program needs to be compiled, a few version of it are generated, using random sets of optimization, and executed. The running times, together with the static features of the code source are used to build a program-specific model. Using this model, the speedup times of all the unseen sets of optimizations can be predicted. The iterative algorithm uses the model to predict the speedup of 500 optimization sets. Then, up to 100 of the best of them are actually used to generate candidate programs to be tested to find the best one. The experiments presented in the paper show a varying number of programs being used to train the model: from 8 to 512. Therefore, the actual number of versions of the program that have to be compiled and executed for every compilation process is fairly high, even if the approach is interesting from a theoretical point of view.

Later, Dubach et al. [34] also presented a technique to allow the compiler to adapt to architectural changes, to find the right optimizations across programs and architectures, without the need to perform a new training every time the architecture changes. Only a single, initial training phase is needed: some sets of optimizations are executed on various program/architecture pairs. Their interactions are characterized using performance counters. Even if this training is quite long, it needs to be performed just once: its results are then usable on a wide variety of new, unseen architetures and programs, therefore its cost becomes negligible. The model that is constructed consists of a mapping from the features of the programs to a probability distribution over good optimization phases. When a new program needs to be compiled, the best predicted executable is immediately generated, without iterative compilation and without the need for a training phase specific for the target architecture. The input required to perform each compilation is the source code of the program to compile, a description of the machine it has to be optimized for, and the performance counters computed by a single profiling run of the program itself, compiled with the default optimization level.

The state of the art in iterative compilation based on machine learning is Milepost GCC [45]. It is a "modular, extensible, self-tuning optimization infrastructure to automatically learn the best optimizations across multiple programs and architectures based on the correlation between [static] program features, run-time behavior and optimizations", and it aims at being the first practical attempt at bringing iterative compilation and machine learning techniques into production compilers. It is built on top of GCC and it allows multi-objective optimization, in order to optimize the code for speedup, code size, and compilation times. It uses offline machine learning, so we can distinguish a training phase (where the model used to make the prediction is built) from a deployment phase, where the model is used (with k-nearest neighbor classification) to decide which optimizations to apply to the programs being compiled. In particular, [45] presents two models that can be used for the learning process. The first one is a probabilistic model used to make *one-shot* predictions on unseen programs, thus completely avoiding iterative compilation and predicting just the single, supposedly best, optimizations set to use. The model is a mapping from the features of a program to a distribution over good optimizations sets. The second model is a transductive machine learning model, that is a model where optimization combinations themselves are used as features for the learning algorithm together with program features. This model aims at predicting whether a combination of program features and optimization passes will give a speedup of at least 95% of the maximal speedup. The probabilistic model is shown to be better at improving the speed of programs.

Although most machine learning algorithms can be classified as offline, there are also some online machine learning algorithms, that will be presented in the next section.

2.3.2 Online machine learning

The traditional machine learning approach deals with using knowledge from the past to make decisions about the present.

An initial training phase is responsible for building a model, and then, during the deployment phase, this model is used to make predictions about new situations.

This approach has two shortcomings: first, the initial training phase is frequently extremely long, therefore limiting the applicability of these techniques. Second, the effectiveness of the model is strongly influenced by the choice of the training set.

To face these issues, a new category of machine learning algorithms, namely *online machine learning algorithms*, has been introduced.

Online learning algorithms are not limited to using data acquired in an initial phase, but keep learning from the data that are used along all the life of the application itself.

The *Long-term learning* algorithm for compilers presented in Chapter 3 is an online learning algorithm.

A good, altough old, survey of this field is [14], by Blum. It describes a few algorithms that can be used to constantly widen the available knowledge and improve the quality of the predictions.

Online algorithms have not been used much in the field of compilers: usually, either the learning process takes place during the initial training phase, or it happens while the program is being compiled by means of iterative compilation. Though, this means that there is no model being actually learned online: only that best candidate is being chosen, without retaining any actual knowledge for the compiler to reuse.

One of the main example I could find of a compiler optimization method based on online learning is described in [84]. The author of the paper uses a Reverse k-Nearest Neighbour classifier and the features of the program being compiled to exploit the knowledge about previously compiled programs to find out what the best set of optimzations to apply is. If no previously compiled program is similar enough to the one currently being analyzed, the system switches to using a random search approach. The program is compiled and run using many different optimization configurations. The results are stored in the database used by the classifier to improve it for the next programs. If the classifier is able to find a good fit, the optimization configuration is used to immediately build the final executable, and the compilation process is extremely fast. On the other hand, if no good candidate can be found, the random search take a really long time, as in the basic iterative compilation approaches.

The IID model presented in [2] can be considered an early example of online learning in compilers.

The work presented in [38] presents an algorithm that can be considered online machine learning, because it focuses the selection of the next candidates by learning as it tests them, but actually only uses the knowledge taken from multiple compilation of the same executable. Therefore it is different from the long-term learning algorithm that we will present in the next chapter. Furthermore, even if the obtained results are really good from the point of view of the obtained speedup, it still needs hundreds of compilation of the single program, taking up to a few hours, as stated in the experimental results section of the paper. This is perfectly fine for tuning software for embedded systems, but has to be reduced in order to obtain a more general usage of learning algorithms for compilers.

Milepost GCC, already described in the previous section, is also part of a bigger project aiming at using online learning and a collaborative enterprise, called *collective optimization* [47], to speed up the acquisition of the training data needed to use a machine learning approach without the need for long training phases and iterative compilation. In this setup, data from many users compiling different programs on different architectures are gathered on a single, public database. One of the main challenges presented by this approach is the need to learn across different datasets, programs and platforms at the same time. Static function cloning and statistical competition between pairs of sets of optimizations provide an efficient method to characterize them with a single run, without using reference runs to calculate the speedups. Collective optimization is performed by computing three probability distributions used for three "maturation" stages of the programs. Programs in stage 3 are well known and heavily used. Enough information has been acquired across datasets to build a program-specific probability distribution. Programs in stage 2 are known, but only a few runs have been recorded. The probability distribution derives from information about other programs behaving similarly (they behave alike for the sets of optimizations already tested). Programs in stage 1 are still unknown. A probability distribution computed as the unweighted average of all the available stage 3 distributions is used for programs in this stage. Still, in order to improve multiple programs across multiple dataset over the optimization level of GCC's `-O3` a few hundred runs (that may be split across multiple users) are needed before a program can reach stage 3. This method has to be considered a form of online learning, because the knowledge base is continuously improved by the data gathered by the users. My work, presented in the next chapter, will improve over the capabilities of Collective Optimization, by having comparable learning times and performance, but increasing the readability of the model (sets of simple mathematical formulas as heuristics, instead of a relatively obscure statistical model) and for the ability to actually consider sets of heuristics as a whole, instead of just considering and evolving the single heuristics on their own.

2.4 Just-In-Time compilation

Sometimes, in order to optimize an application to run on the current system, modifying it at compile time using the approaches presented in Sections 2.2 and 2.3 is not enough. Certain parameters of the program might need to be modified at runtime, exploiting the knowledge about the specific dataset being elaborated.

The problem of adapting programs to the runtime environment and to the specific set of data they are working on has been tackled in many ways, mostly related to the concept of dynamic compilation, also known as Just-In-Time (JIT) [10] compilation. According to this approach, parts of a program are compiled while the program itself is being run, when more information is available and can be used to perform further optimizations.

One of the first works on JIT systems [55] deals with the fundamental questions of JIT: determining what code should be optimized, when, and which optimizations should be used.

JIT compilation introduces an overhead in execution time because it causes the program to be idle while waiting for the new machine code. Considering that most programs spend the majority of time executing a minority of code [71], two papers [25, 28] independently proposed the approach called *mixed code*, where most of the code is interpreted and only the frequently executed part is identified, compiled and optimized at runtime.

Some works [57, 76] exploit multi-core processors to hide compilation latency: the compiler is run in a different thread and uses heuristics to predict the next method to compile before it is actually needed by the program. State of the art implementation can be found in [74].

A similar approach is used by the ILDJIT compiler [18], which I contributed to develop [117, 118] before focusing on the topic of this thesis work.

Another insteresting work in the area of JIT compilation is [62], that proposes a technique for dividing the analysis and compilation of programs into phases, anticipating as much analisys as possible at compile time, and generating programs with ”holes” to be filled at runtime after the remaining part of the analysis has been completed.

2.5 Autotuning

The JIT approach is too resource-hungry for many applications, since it requires a full-fledged compiler to be available and running at all times alongside the application itself.

In many cases, autotuning may be a better way to obtain a very similar result. It consists in having the application exposing certain parameters to be tuned to adapt its functioning to the available system resources and input data.

Ansel et al. present PetaBricks [5], a language and compiler enabling the programmers to provide multiple implementations of algorithms to allow the program to decide at runtime which one to use, combining them to form new, more complex, algorithm where possible and when needed. It also allows the program to expose numerical parameters to be tuned to provide the best performance on the current system. The tuning process of PetaBricks is based on the INCREA evolutionary algorithm, detailed in [6].

Another example of autotuning can be found in the works by Tiwari et Al. [122, 123]. The authors aim at building a “general-purpose, offline auto-tuning framework for whole programs [integrating] performance analysis tools, compiler frameworks and autotuners [and at] developing and deploying a fully automated tool-chain that can provide end-to end tuning for full programs”. Their system is based on three components. ROSE is able to extract computational hotspots from large applications into separate functions. These functions are then used by CHiLL, the second component, a polyhedral loop transformation and code generation framework that generates multiple optimized variants of them. Finally, the last component, Active Harmony, is responsible for the actual autotuning phase. It allows to describe and export a set of performance-related tunable parameters. Then, it uses a parallel search algorithm (PRO: Parallel Ranking Order) to leverage parallel architectures to simultaneously search across a set of optimization parmeter values, lookin for the ones that best fit the runtime environment.

Autotuning techniques have been used also in other, more restricted contexts, such as described in [26] where the authors present “an autotuning environment for stencil codes that searches over a set of optimizations and their parameters to minimize runtime and provide performance portaiblity across the breadth of existing and future architectures”.

The portability toward future architectures is one of the major strenghts of autotuning: given that the executable itself is able to adapt to the environment, without the need for a compiler, even compiled programs can continue to be used with good performance on changing hardware platforms.

2.6 General techniques

Certain techniques used in this thesis are not strictly related to the field of compilation, but are of primary importance for the development of the following chapters. This section will briefly present them.

2.6.1 Evolutionary algorithms

Many different techniques can be used to build the model used for machine learning approaches. In this section I will briefly present those based on evolutionary algorithms, because they are the basis upon which Chapter 3 is developed.

Evolutionary algorithms [37] aim at exploring a solution search space looking for the best solution of the problem at hand by mimicking the way biological evolution works, exploiting the computational power of modern computer to try out many different solutions, though without the need of performing an extremely time-consuming exaustive search of the whole search space.

At the basis of an evolutionary algorithm is a population composed by a series of individuals. Each of the individuals represent a possible solution to the problem that the algorithm aims to solve.

Evolutionary algorithms have a fitness function that evaluates how good a solution each individual of the population is. Only the individuals determined to be the fittest survive.

Each new generation is composed by a new set of individuals. The new individual are created by evolving the fittest individuals that survived the previous generation. This works under a locality assumption, that is, if it is true that by applying minor modification to an individual of the population its performance changes marginally.

When this is true, the evolution step can actually improve the population each generation, by exploring the surrounding of the search space starting from those points that are already known to perform well because they represent individuals that have been proven to have a high fitness value. Therefore, we can say that evolutionary algorithms are a biased sampling search technique.

Clearly, the exact encoding that represents an individual depends on the specific problem that the evolutionary algorithm aims to solve. The creation of the individuals of the new generation is also problem dependent, but we can identify a few characteristics that can be found in many evolutionary algorithms.

Evolution This is the core of the algorithm itself. It takes one of the best candidates of the previous generation and slightly modifies it, creating a new candidate. In most algorithms the modification is completely random, whereas in some implementation it is guided to prefer certain solutions over others.

Mutation The population of the first generation is usually created randomly. This means that it biases the solution found by the next generations, leading them to explore around the area of the search space around the spots were the individuals of the first population were. Mutation is meant to solve this problem. It is activated with a low probability. When it is activated, it heavily modifies one candidate, or generates a completely random new one. This way, it prevents the evolution process from being stuck in local minima.

The general structure of an evolutionary algorithm is represented in Figure 2.1

Evolutionary algorithms can be used in many different areas because they do not impose many constraints on the tractable problems: they just require that given two individuals representing candidate solutions, it is possible to determine which one is the best.

They are also particularly useful in case when the availability of new data requires the selection of the best candidate to be performed again. The latest best population known can be used as the starting point of a new evolution process, aimed at fitting the new data and the new available knowledge. This is why in Chapter 3 I used an evolutionary approach as the basis of an algorithm for optimizing a compiler to improve its ability to produce good programs for its target architecture.

Genetic algorithms

Genetic algorithms are a particular kind of evolutionary algorithms. They are meant to imitate the natural evolution process even more closely, because they are built to mimick the process of reproduction.

In genetic algorithms the encoding representing each individual of the population is called a *gene*.

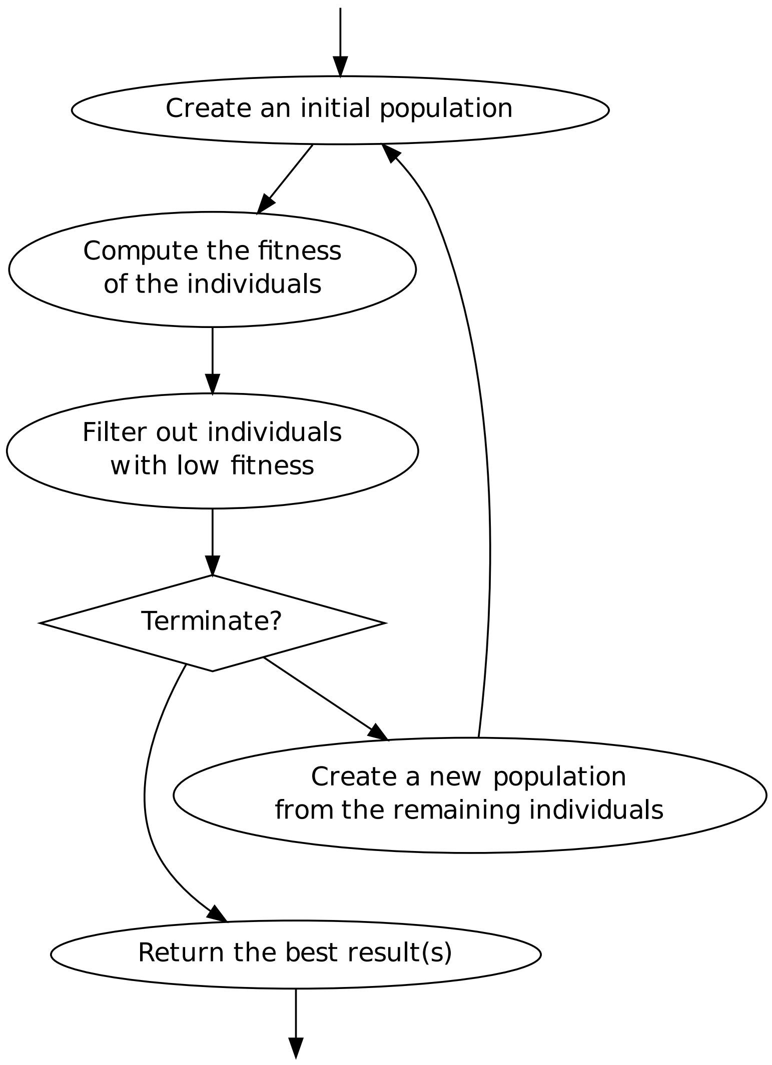

Create an initial population
Compute the fitness of the individuals
Filter out individuals with low fitness
Terminate?
Create a new population from the remaining individuals
Return the best result(s)

Figure 2.1: The structure of an evolutionary algorithm

In genetic algorithms, with respect to generic evolutionary algorithms, a new evolution operator is added, called *cross-over*. The cross-over operator works by considering two parent individuals instead of just one, while generating the individuals of the next generation population. It takes two individuals that scored high according to the fitness function, selects a cutting point in their genes and crosses the resulting fragments, thus obtaining two new individuals, each with about half the genetic code of each of the two parents.

The core idea behind cross-over is that by taking the genetic code of good individuals, the offspring is likely to yield good results. At the same time, taking code from two different individuals helps reducing the possibility of getting stuck in local minima.

Genetic algorithms are frequently used for machine learning-based compilation, so it is worth mentioning them. Though, they are not used for the work I present in Chapter 3, because the reduced number of individuals used for each generation in such work makes an algorithms without cross-over more practical for our purposes.

Genetic programming In genetic algorithms, an individual can be anything that can be represented in the gene.

Using an appropriate encoding, it is possible to represent a program inside a gene (usually as a linearization of its abstract syntax tree), and therefore it is possible to have the genetic algorithm evolve a program.

This technique is known as genetic programming [75].

The main restriction of genetic programming is due to the fact that, being a genetic algorithm, it uses crossover. Therefore all the nodes of the tree (that is, the program) need to be compatible, that is, they need to return the same kind of data.

Grammatical evolution Grammatical evolution [108] was introduced in order to solve the shortcoming of genetic programming.

By coupling a grammar in Backus-Naur Form[70] grammar to the genetic programming algorithm, it is possible to have each node of the program have a different type. It will be the grammar's responsibility to define which kind of node can be generated in every point of the gene.

In Chapter 3 I use an approach similar to grammatical evolution, based on a grammar but without the cross-over operator, to guide the creation of heuristics for an optimizing compiler.

2.6.2 MapReduce

MapReduce is a programming model that allows the programmer to expose the concurrency of programs in an easy way, and it makes it easy to process large data sets, usually by using distributed computation exploiting the computational power of clusters of computers.

The name comes from the two main functions that have to be implemented by the programmer using this programming model, namely *Map* and *Reduce*. This terminology comes from the world of functional programming languages.

The *Map* function applies the same computation to all the elements of the data set and returns a series of pairs (one for each input) including a key and a result.

The *Reduce* function, later, takes all the data produced by the Map function having the same key and iterates over them to compute a final aggregated results for each key.

The full MapReduce model also includes other functions, but they are used mainly as helpers to increase the efficiency or the efficacy of Map and Reduce, so they will not be described here.

Since its introduction [29] by Google, the MapReduce parallel programming model has been widely adopted in many different fields, because of its high scalability for applications where many independent jobs have to be processed. Many implementations are available, Hadoop [120] being the most famous and widely used of those publicly available.

Most implementations of MapReduce are meant to be used on clusters of machines for high performance computing tasks. Nevertheless, MapReduce can also be useful as a programming model for applications that can be conveniently expressed in a concurrent way, even when they do not require high computational power and are meant to be executed on a single desktop computer: Ranger et al. present Phoenix [105], an implementation of MapReduce for multi-core and multiprocessor systems. Later, Yoo et al. [132] further improved the system and Talbot et al. rewrote it from scratch to remove some limitations and inefficiencies, presenting Phoenix++ [115], featuring greater scalability and much better performance.

MapReduce has been used for easily adding concurrency to many different tasks related to a wide variety of fields. Particularly interesting for our purpose is the work by Gillick et Al. [49]. The authors show that MapReduce can be profitable for offline machine learning applications, where all the training runs are executed first, to learn the model that is later used. In Chapter 4 I will present a technique to execute timing runs for a machine learning-enabled compiler in parallel, using MapReduce inside an online machine learning algorithm, where learning takes place at the same time when the model is used.

2.6.3 Principal Component Analysis

When using machine learning it is of paramount importance to correctly identify the features that will convey the most information about what is being learned, while at the same time being a really restricted number. This is due to the fact that for many learning algorithms it is easier to deal with fewer input data.

Most of the works presented in Section 2.3 try to reduce the size of the input space by using a technique called Principal Component Analysis (PCA) [130]. PCA uses a linear transformation that projects the available variables onto a new set of orthogonal axes. The new variable with the highest variance is projected on the first axis, the variable with the second-highest variance on the second axis and so on.

By keeping the first n variables of the transformed system, you can preserve most of the information of the original data while reducing the number of variables needed to represent them.

In machine learning compilers, PCA is usually used on the set of features extracted from the training programs. Instead of storing all the features of the programs, they are first transformed using PCA. Usually, different features are correlated, and storing the original data "as is" would lead to duplicate information. PCA allows to factor out the information into a small set of variables.

Principal Component Analysis, in its traditional form, requires all the data to be available before it can be applied. Therefore, it can be used only for offline learning algorithm.

Recently, though, *Incremental Principal Component Analysis* was introduced and used in various fields [8, 24, 95], such as computer vision. This techique is meant to behave as traditional PCA, but without the need for having all of the data available beforehand.

A similar technique could therefore be used for online machine learning.

3 Long-term Learning of Compiler Heuristics

[1] Optimizing programs to exploit the underlying hardware architecture is an important task. Much research has been done on enabling compilers to find the best set of code optimizations that can build the fastest and less resource-hungry executable for a given program. A common approach is iterative compilation, sometimes enriched by machine learning techniques. This provides good results, but requires extremely long compilation times and an initial training phase lasting even for days or weeks.

We present long-term learning, a new algorithm that allows the compiler user to improve the performance of compiled programs with reduced compilation times with respect to iterative compilation, and without an initial training phase. Our algorithm does not just build good programs: it acquires knowledge every time a program is compiled and it uses such knowledge to learn compiler heuristics, without the need for an expert to manually define them. The heuristics are evolved during every compilation, by evaluating their effect on the generated programs. This is similar to what [47] does, but it is more powerful, because multiple heuristics are evolved together, as sets, to ensure that they interact

[1]This chapter, with the title "Continuous Learning of compiler Heuristics", has been accepted for publication in the ACM Transactions on Architecture and Code Optimization and for presentation at the 8th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'13), January 21-23, 2013, Berlin, Germany

correctly with one another. Furthermore, we evolve readable heuristics instead of just a statistical model.

We present implementations of long-term learning on top of two different compilers, and experimental data gathered on multiple hardware configurations showing its effectiveness.

The data show that long-term learning is able to improve the performance of a compiler over time, without the need for any initial training phase.

We also show that different compilers can benefit from the use of long-term learning, by presenting two implementation and experimental results gathered on multiple hardware configurations.

The rest of this chapter is organized as follows: Section 3.1 describes the long-term learning compilation algorithm, Section 3.2 describes our implementations of the algorithm and the setup of the experimental campaign, Section 3.3 presents the results of the experiments. Related work is presented in Section 3.4 and possible future improvements are discussed in Section 3.5. Section 3.6 concludes.

3.1 Long-term learning compilation

Long-term learning compilation is a new continuous learning algorithm meant to avoid the main drawbacks of both iterative and machine-learning-based compilation: it only needs a small number of compilations and test runs to be performed for each program, thus reducing compilation times with respect to iterative compilation; at the same time, it does not need the initial training phase usually required by machine learning algorithms.

The aim of long-term learning is to learn a model of the target architecture, in the form of a set of heuristics that will allow the compiler to produce highly optimized programs for that target architecture, without manual tuning by an expert. This learning process takes place over time: every time a new program is compiled, the system learns some new piece of information to be reused during the next compilations, therefore we say long-term learning is an *online* learning algorithm. [14]. *Offline* learning algorithms, on the other hand, have separate training and working phases. This is possible thanks to the testing phase of the various candidate versions of the program, that happens as a part of the compilation process.

Long-term learning (detailed in Algorithm 4.2.1) is an *evolutionary algorithm* [37].

As such, it mimics biological evolution to solve the problem of finding the best set of compiler heuristics. Each set of compiler heuristics (also termed *candidate compiler configuration*) is seen as an individual of a population. The population evolves along a given number of generations, each composed by the same number of individuals. The individuals of each generation derive from the best ones of the previous generation by modifying them slightly through a process named *evolution* (described in Section 3.1.1). At the end of each generation, a *fitness function* evaluates each individual according to the method explained in Section 3.1.6: the fittest individuals are then used as the seed to generate the candidates of the next generation.

In order to prevent stagnation of the process into local performance maxima, evolutionary algorithms also define a *mutation* operation (detailed in Section 3.1.2), that modifies the candidate configurations more deeply than evolution does, allowing to look for solutions that are far away from the current best ones in the search space. To speed up the convergence of evolutionary algorithms and to improve the stability of the solution found, a technique known as *elitism* (described in Section 3.1.4) can be used: elitism copies the best candidates found by the previous generation in the new one, without any modification.

In our system, we decided to represent each compiler heuristic contained in the candidate configurations as a mathematical formula (such as those in Figure 3.1 and in Section 3.3.3). The result value of the formula is the decision made by the heuristic, and variables of the formula represent *features* of the program being compiled. A feature is a value known by the compiler at compile time that describes a characteristic of the program (*e.g.* number of basic blocks in a function, loop nesting level, *etc.*). A similar representation of heuristics has been used by [113], although for a system based on offline learning. In particular, we use static features: features that can be extracted by source code analysis, without requiring the program to be executed.

The heuristics are not hard-coded into the compiler source code, to allow them to evolve easily: every time the compiler needs to make a decision, it loads the current best heuristic for that decision from a knowledge base, evaluates the heuristic using the values of the features and uses the obtained result as the outcome of the decision process.

In most evolutionary algorithms the individuals of each generation descend from the individuals of the previous generation only. Not so in long-term learning: at the end of every generation, all the available information is stored in a knowledge base (see Section 3.1.5), and the new candidates are evolved from the best ones of the whole knowledge base. Moreover, we do not just store information about the best candidates, but about all of them, together with their fitness level describing how

Boolean formula $true$

Boolean formula $if((avg\_basic\_block\_less\_15\_inst < 11))\ then\ (true)$
$else\ (avg\_basic\_block\_two\_successors > 10)$

Double formula $(avg\_method\_assignment\_inst/3)$

Figure 3.1: Examples of formulas generated as heuristics by the long-term learning algorithm.

good each heuristic has proved to be over the whole life of the compiler. We can thus reduce the number of generations needed to obtain good candidates, since we are using all the already gathered data instead of partial information. The heuristics stored in the knowledge base (and in particular, the best-scoring ones) constitute the model of the target system that the algorithm learns over time.

We determined experimentally that our algorithm performs well with just 3 generations and 6 candidates (including the elite) per generation. Therefore, with less than 20 candidate configurations tested during each compilation we can obtain good results without the need for an initial training phase, whereas other iterative approaches need hundreds of them. It should be noted that using 6 candidates over 3 generations leads to better results than just compiling 18 candidates in a single generation. This is due to the fact that using a single generation, all the candidates are randomly generated starting from the pre-existing knowledge of the compiler, whereas using multiple generations the candidates of each generation are based upon the knowledge acquired during the previous one for the specific program being compiled.

It is also worth noting that we chose not to implement a genetic algorithm because that would make more difficult to use a reduced number of candidates. Genetic algorithms require that each candidate descends from two candidates of the previous generation, by applying a crossover operator that mixes the genes of the two candidates. In order for this to perform at its best, a large population is required. On the other hand, by developing an evolutionary algorithm that is not genetic, we could devise a different evolution method, starting from the knowledge contained in the whole knowledge base, thus allowing us to use less candidates in each generation.

The general workflow of a compiler using long-term learning is represented in Figure 3.2.

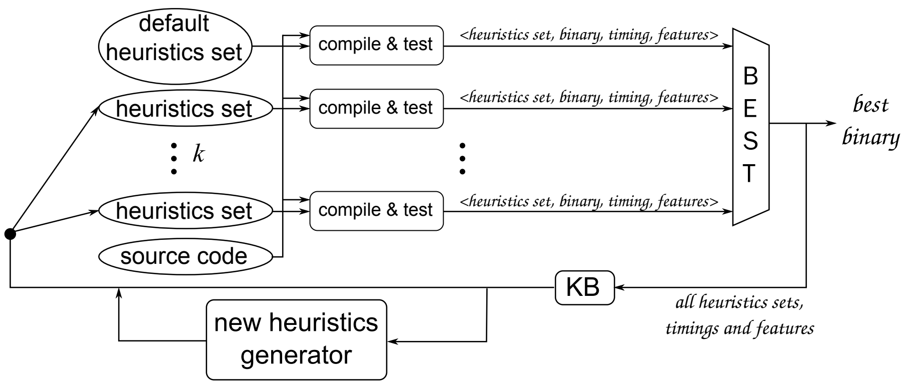

default heuristics set
heuristics set
k
heuristics set
source code
compile & test
compile & test
compile & test
<heuristics set, binary, timing, features>
<heuristics set, binary, timing, features>
<heuristics set, binary, timing, features>
BEST
best binary
KB
all heuristics sets, timings and features
new heuristics generator

Figure 3.2: The workflow of a compiler using long-term learning. The number of candidates to test is k.

3.1.1 Evolution

In order to learn, every evolutionary algorithm needs to explore the solution space by generating and testing new candidates. As the name "evolution" suggests, new candidates are derived from the old ones in a non-disruptive way. The idea is to start from a candidate configuration known to be good, and to modify it slightly, looking for a new configuration that yields better performance when applied.

First of all, the GetBestCandidates function of Algorithm 3.1.1 extracts from the knowledge base as many candidates as required to reach the user-specified number k of candidates to test. They are taken from the fittest candidates (according to the fitness function described in Section 3.1.6) known up to the current time. Each heuristic is associated with a decision point name that univocally identifies in which phase of the compiler it has to be used. Every time the compiler needs to make a decision, it picks from its configuration the unique heuristic formula corresponding to that decision point. Therefore, each configuration must contain exactly one formula for each decision point name. The set of all possible names is denoted by N in Algorithm 3.1.1. Let H_i be the set of the names of the heuristics contained in candidate i extracted from the knowledge base. In case H_i is not equal to N, the missing heuristics (identified by the names in $N \setminus H_i$) will be randomly generated by the GenerateMissingHeuristics function and used to complete the candidate just before testing it. This happens in particular when not enough candidates where found in the knowledge base: they are replaced by empty candidates, to be later filled in by GenerateMissingHeuristics.

The GetBestCandidates function ensures that

$$\forall i \quad H_i \subseteq N \tag{3.1}$$

Such property ensures us that the heuristics in the set are known to work well together. By considering supersets too, it might happen that the extra elements (to be discarded since they are not needed) are essential to deliver the performance level promised by the score of the set of heuristics. This would lead to generating a candidate with a poorly performing set of heuristics.

After the candidates have been extracted from the knowledge base, on each of them we execute the Evolve function with a given probability (p in Algorithm 3.1.1), meant to model the *exploration* versus *exploitation* tradeoff [61], that is the expected utility of trying new solutions instead of reusing the old ones, proven good. When the exploration decision is taken, Evolve determines the number of heuristics to evolve, the minimum being one and the maximum being a percentage e of them. Then, the actual evolution process takes place for randomly chosen heuristics of the set. We allow more than one heuristic to evolve at once to enable a group of decisions that are effective only when applied together to be made at the same time.

When a formula is being evolved, it is parsed according to the grammar in Figure 3.3, obtaining a syntax tree where every leaf represents a numeric or boolean value or a feature name, and every internal node represents an operator or an *if* expression. Then, the tree is recursively visited, until a single terminal of the grammar is actually modified according to the following rules:

If expression (If$_t$) Evolution is recursively applied to one element chosen randomly between the condition formula, the *then* case formula and the *else* case formula.

Binary operation (BinOp$_t$) Evolution is applied to one element chosen randomly between the left operand, the right operand and the operator. In the first two cases, the evolution process continues recursively, in the last case the operator is substituted with a different one taken from the same category (Arith, Bool or Comp).

Boolean value ($Boolean$) The value is negated.

Feature ($Feature$) A different feature is used.

$$G = \begin{cases}
\text{S} \rightarrow \textsc{NumFormula} \mid \textsc{Formula}_{bool} \\
\textsc{NumFormula} \rightarrow \textsc{Formula}_{int} \mid \textsc{Formula}_{double} \\
\textsc{Formula}_{int} \rightarrow \textsc{BinOp}_{int} \mid \textsc{If}_{int} \mid \mathit{Integer} \mid \mathit{Feature} \\
\textsc{Formula}_{double} \rightarrow \textsc{BinOp}_{double} \mid \textsc{If}_{double} \mid \mathit{Double} \mid \mathit{Feature} \\
\textsc{Formula}_{bool} \rightarrow \textsc{BinOp}_{bool} \mid \textsc{If}_{bool} \mid \mathit{Boolean} \\
\textsc{BinOp}_{int} \rightarrow \text{“(”}\ \textsc{Formula}_{int}\ \textsc{ArithOperator}\ \textsc{Formula}_{int}\ \text{“)”} \\
\textsc{BinOp}_{double} \rightarrow \text{“(”}\ \textsc{Formula}_{double}\ \textsc{ArithOperator}\ \textsc{Formula}_{double}\ \text{“)”} \\
\textsc{BinOp}_{bool} \rightarrow \text{“(”}\ \textsc{Formula}_{bool}\ \textsc{LogicOperator}\ \textsc{Formula}_{bool}\ \text{“)”} \\
\qquad\qquad \mid \text{“(”}\ \textsc{NumFormula}\ \textsc{ComparisonOperator}\ \textsc{NumFormula}\ \text{“)”} \\
\textsc{ArithOperator} \rightarrow \text{“}+\text{”} \mid \text{“}-\text{”} \mid \text{“}*\text{”} \mid \text{“/”} \\
\textsc{BoolOperator} \rightarrow \text{“}=\text{”} \mid \text{“\#”} \mid \text{“}<\text{”} \mid \text{“}>\text{”} \mid \text{“}<=\text{”} \mid \text{“}>=\text{”} \\
\textsc{ComparisonOperator} \rightarrow \text{“}\mathit{and}\text{”} \mid \text{“}\mathit{or}\text{”} \\
\textsc{If}_{int} \rightarrow \text{“}\mathit{if}\text{”}\ \text{“(”}\textsc{Formula}_{bool}\text{“)”}\ \text{“}\mathit{then}\text{”}\ \text{“(”}\textsc{Formula}_{int}\text{“)”} \\
\qquad \text{“}\mathit{else}\text{”}\ \text{“(”}\textsc{Formula}_{int}\text{“)”} \\
\textsc{If}_{double} \rightarrow \text{“}\mathit{if}\text{”}\ \text{“(”}\textsc{Formula}_{bool}\text{“)”}\ \text{“}\mathit{then}\text{”}\ \text{“(”}\textsc{Formula}_{double}\text{“)”} \\
\qquad\quad \text{“}\mathit{else}\text{”}\ \text{“(”}\textsc{Formula}_{double}\text{“)”} \\
\textsc{If}_{bool} \rightarrow \text{“}\mathit{if}\text{”}\ \text{“(”}\textsc{Formula}_{bool}\text{“)”}\ \text{“}\mathit{then}\text{”}\ \text{“(”}\textsc{Formula}_{bool}\text{“)”} \\
\qquad \text{“}\mathit{else}\text{”}\ \text{“(”}\textsc{Formula}_{bool}\text{“)”}
\end{cases}$$

Figure 3.3: Grammar representing all the formulas that can be used as heuristics. S is the axiom. *Integer* and *Double* represent, respectively, integer values and double precision floating point values. *Feature*s are represented in formulas as variables, each with a unique name. This will be substituted with their actual values by the compiler, when the heuristic is evaluated.

Numeric value (*Integer* or *Double*) The current value v is incremented by a value in the interval $[-v, v]$. The resulting value can only be in the interval $[0, 2v]$ if $v > 0$ or $[2v, 0]$ if $v < 0$. This is done to prevent the value from changing too much, under the assumption that a heuristic being evolved derives from one already evaluated as good, and therefore does not need to be drastically changed.

The main difference between this and more traditional evolutionary approaches applied to compilers is in the target of the evolution process. Usually, the process aims at finding a good candidate configuration, that is a set of compilation flag and parameters to be applied to the current program in order to speed it up. In the case of long-term learning, on the other hand, a candidate is not a compiler configuration but a set of heuristics that the compiler can use to determine, on a program-by-program basis, what the best compilation options and parameters are. Therefore, the produced results are much more general: we are not just learning how to deal with a single program, but we are actually learning a model that behaves increasingly well with all the programs it interacts with, and that can be used with unchanged efficiency even if the learning algorithm itself is later disabled.

3.1.2 Mutation

Even if we modify multiple heuristics at once, there is a risk of getting stuck in local maxima, since evolution only applies small modifications to formulas. To avoid this, we implemented a mechanism able to provide bigger evolutionary steps, traditionally named mutation [107]: every time the evolution function operates on a node of the syntax tree of a heuristic formula, the evolution might be turned into a mutation with probability m.

The conceptual difference between evolution and mutation is in the editing distance. Evolution recursively visits the tree until it modifies a single terminal of the grammar. Mutation can be applied to either a terminal or a non-terminal and is able to change the formula much more deeply. The modifications applied to the formula are as follows:

Whole formula (S) The formula is substituted by a completely new one.

If expression (If_t) Either the condition, the *then* case formula or the *else* case formula is removed and substituted with a new sub-formula, generated from scratch.

Binary operation (BinOp$_t$) Either the left or right operand sub-formula is substituted with a new sub-formula, generated using the grammar in Figure 3.3.

Boolean value (*Boolean*) Same behavior as evolution: the current value is negated.

Feature (*Feature*) Same behaviour as evolution: a different feature is used.

Numeric value (*Integer* or *Double*) A random number is generated. It can be unbounded or, where specified, comprised in an interval determined by the validity range (see Paragraph "Validity range check" of Section 3.1.3) of the heuristic itself.

After the mutation takes place, the formula is turned back into its string representation inside the heuristic.

3.1.3 Generation of new formulas

All the formulas that can be used as heuristics in our system can be described using the grammar in Figure 3.3. Every time a formula needs to be generated, the grammar is used as a generative device. The generation starts from axiom S. The first choice (between FORMULA$_{bool}$ and NUMFORMULA) depends on the specific type of result that is needed. If the heuristic aims to decide whether or not to activate an optimization, a BOOLFORMULA will be generated. If a compiler parameter has to be heuristically determined, NUMFORMULA will be used, in one of its two versions (integer or double).

The choice of the right grammar is essential to generate good heuristics. We want to give the system enough flexibility to find good results. At the same time, as stated in [113], the wider the heuristic search space, the longer it will take for the algorithm to converge upon a general solution. Any grammar could be expanded to contain more primitives able to produce more complex formulas, so it will never be possible to generate a completely unbiased set of heuristic formulas. Since we are defining an online learning system, it is particularly important for it to converge rapidly, so we choose to only use basic arithmetic and logic operands for the formulas. Machine learning techniques aim at being faster than iterative ones by biasing the search space in some way, and experimental results on multiple platforms and multiple test programs have proved our choice to be valid. Future work could look for more complex primitives able to deliver significant performance improvements.

$$\textsc{Formula}_t \rightarrow \begin{array}{ll} \textsc{BinOp}_t & \mathit{weight}: 2 \\ \mid \textsc{If}_t & \mathit{weight}: 1 \\ \mid \mathit{FormulaType}_t & \mathit{weight}: 5 \\ \mid \mathit{Feature} & \mathit{weight}: 4 \text{ (only if } t \text{ is } \mathit{int} \text{ or } \mathit{double}) \end{array}$$

Figure 3.4: The weights applied to the grammar to limit the expansion of heuristic formulas. $\mathit{FormulaType}_t$ is respectively *Integer*, *Double* or *Boolean* for t equal to *int*, *double* or *bool*.

Preventing indefinite growth of heuristics

Formulas generated by the grammar are defined recursively and could be arbitrarily long. Should the grammar be used with uniform probability of choosing any of the alternative productions, we would obtain complex heuristics. The length of formulas would tend to increase rapidly, making them slower to evaluate and harder to read and understand. Generating readable heuristics is not strictly required, but can be really useful for compiler writers. When developing new optimizations, being able to look at a heuristic and understanding how it makes decisions is handier than systems where the optimization decision process is less visible, such as the ones in [84, 45].

The importance of preventing a grammar from generating too long formulas is described in [81, 80]. They use a grammar to automatically define complex features to be extracted from programs and used in a classifier for implementing a machine-learning based compilation algorithm. We use a similar approch, based on assigning weights to the various productions of the rules, but in our algorithm the set of features is fixed and we use the grammar to generate the heuristics the compiler will use to make decisions. In our specific case, weights are needed only for the rules describing formulas (*i.e.* $\textsc{Formula}_{int}$, $\textsc{Formula}_{double}$ and $\textsc{Formula}_{bool}$). The weights we use are described in Figure 3.4. The actual production to activate is chosen by a random roulette-wheel selection algorithm[2].

For every other rule with more than one production ($\textsc{BinOp}_{bool}$, ArithOperator, BoolOperator, ComparisonOperator) the selection is performed with a uniform probability, since all the alternatives generate formulas of equivalent length.

[2]Given a set of elements, each with a fitness value, a roulette-wheel selection algorithm chooses an element from the set with a probability proportional to the fitness of the element itself. We use weights as the fitness value.

When the $Feature$ production is activated, the learning algorithm randomly chooses a feature from the set F of the features whose value will be provided at compile time.

Reducing the search space size

The search space of possible heuristics described by Figure 3.3 is infinite. The long-term learning evolutionary algorithm explores it efficiently, exploiting the acquired knowledge to avoid testing candidates likely to perform poorly, and focusing on evolving the sets of heuristics that have proven to be beneficial to the execution times of previously compiled programs.

Every restriction of the search space helps to speed up the convergence toward a useful solution. Therefore, we use two techniques to achieve this.

Constant folding For every heuristic generated by the system, we apply constant folding techniques (*e.g.*, if a fragment of a heuristic formula is $2 * 3 + 4$ or $1 + 2 + 3 + 4$ it will be folded to 10 in both cases) before storing the heuristic in the knowledge base.

This way, apparently different heuristics are reduced to a single one and the information gathered from those heuristics will converge into a single knowledge base record, making the data about that heuristic more complete and more useful. Having fewer heuristics makes evolution more efficient, since there are less starting points to use. Furthermore, this leads to simpler heuristics being stored.

Validity range check The second technique we use is applicable only to non-boolean heuristics and is implemented as part of the scoring system (see Section 3.1.6). It has been introduced after observing that most numerical heuristics must generate results within a given range of values. For example, a heuristic determining the unrolling factor of a loop has 1 as the minimum acceptable value (meaning no unrolling) and could have a maximum around 100 (or it could also be left unbounded). A heuristic determining the number of worker threads a program should launch could be limited upwards by the number of cores actually available on the machine.

To take into account such boundaries, we introduced the possibility to specify a minumum value min and a maximum value max for every heuristic. The formulas we use are randomly generated, and they include as variables the features extracted by the compiler, therefore it is possible that their evaluation might result in a value outside the valid range. The trivial solution is substituting values below *min* with *min* itself and values above *max* with *max* itself. Such an approach ensures valid results, but impacts negatively on the heuristics: if the value producing the optimal executable corresponds to the *min* of the range, every formula evaluating to something below *min* would receive a high score. This is bad, because it would increase the number of formulas that are considered good, increasing the number of seemingly good starting points for the next generations. We decided instead to add a penalty (precisely described in Section 3.1.6) for formulas evaluated outside the valid range. Therefore, their score will become lower, favoring only the heuristics that stay within the boundaries.

In order to further reduce the possibility of generating low-scoring formulas, the *min* and *max* values are provided to the formula generation function, that exploits them to build formulas that cannot, by construction, be evaluated outside the validity range, as far as no variables (that is, features) are concerned.

Principal Component Analysis As described in Section 2.6.3, many compilation approaches based on machine learning techniques use Principal Component Analysis to reduce the number of input variables used to train the model.

As reported in most of the previous works listed in Section 2.3, whatever the selection of the program features, it is always likely that part of the information contained in each feature is superfluous because other features contain that same information too.

Because of this, we considered using Principal Component Analysis (PCA) for our work. Unfortunately, PCA requires all the data to be available in advance, therefore it conflicts with the core principle of this work of not needing any training phase.

Bibliographic research led us to find out about Incremental Principal Component Analysis, being used in various fields [8, 24, 95] and able to extract a reduced set of features containing the maximum amount of information in as few variables as possible, in such a way that each variable is orthogonal to every other variable.

We considered using incremental PCA to reduce the amount of features we learn from, but in the end we decided not to use it: incremental PCA ensures that a set of features of the desired size is used, but doesn't tell us anything about the meaning of each of the variables. When new data arrive, the content of the new variables might be quite different from the previous ones, because the linear transformation creating the variables changes every time new data are available.

Given that we use the variables as components of formulas explicitly describing the heuristics to be used, changing meaning of the underlying variables every time new data are available (that is, every time a new program is compiled) would lead to an unstable system, increasing the difficulty of identifying good heuristics instead of simplifying it.

3.1.4 Elitism

If all the candidates in a generation are randomly generated, it might happen that their results are worse than the one obtainable by using the best set of heuristics known up to this point. Elitism is a common solution applied by evolutionary algorithms to this problem: it improves the stability of the learning process. Long-term learning uses a very reduced amount of candidates, so rapidly reaching stability is particularly important. Therefore, we implemented three different elitism mechanisms.

The GetBestCandidates function of Algorithm 3.1.1 (already described in Section 3.1.1) extracts from the knowledge base b candidates whose heuristic formulas have been proved to be the highest-scoring ones when used as sets.

GenerateCandidatesFromBestHeuristics extracts from the knowledge base, for each needed heuristic name $n \in N$, the h highest-scoring heuristics. Then, combines them in sets, each containing one heuristic for each needed name. These sets are returned as candidates. This is done because heuristics receiving high scores independently from the specific heuristic set they have been used in, are likely to receive a high score also when introduced in a new set.

GenerateCandidatesFromMostFrequentHeuristics extracts from the knowledge base, for each heuristic name $n \in N$, the f most frequently used heuristics whose score is at least 1 (meaning they provide no slowdown, as described in the next section). Then it combines them in sets, each containing one heuristic for each name. These sets are returned as candidates. This is done because if an heuristics is frequently used without its score becoming too low, it means that, on average, the heuristic is able to provide acceptable performance. Reusing such an heuristic again provides a good fallback candidate, increasing the stability of the algorithm.

By analyzing the learning algorithm it is possible to determine why it could be slow to converge. Each of the elitism mechanisms aims at removing one cause of instability. The first by keeping the current best results across generations. The second one by forcing the creation of candidates likely to perform well. The third one by providing an acceptable fallback option in case every other candidate is not good enough. The third one is especially useful at the beginning of the learning process, where it might happen that best set of heuristics performs well for all the programs compiled up to now, but still fails on a new program, too different from the previous ones.

3.1.5 Knowledge base

The knowledge base stores information about all the tested candidates. In particular, for every heuristic, the knowledge base contains the name of the decision point of the compiler the heuristic will be used for, the formula of the heuristic, the score earned by the heuristic up to the current time and the number of previous uses of the heuristics.

The knowledge base also stores all the sets each heuristics has been part of and, for each set, its score and its number of previous uses.

3.1.6 Computing the fitness function and updating the knowledge base

For each candidate i, we record the execution time $execTime_i$ and we compute the $speedUp = \frac{execTime_0}{execTime_i}$ with respect to the execution time $execTime_0$ of the default configuration $\overline{C}$. Speedups greater than 1 indicate candidates actually providing a performance improvement. If $speedUp > 1$, the candidate provides a performance improvement.

The UpdateKnowledgeBase function uses the speedups as the fitness function of the heuristics and updates the scores contained in the knowledge base.

Every candidate contains a set of heuristics, each controlling a compiler decision. We keep track of how well the heuristics of the candidate work together by computing the score of the set of heuristics as a whole. At the same time, we keep a score for every single heuristic, to know how well it works whatever heuristic set it is part of.

We also keep a use count for every heuristic set and for every heuristic. A set is counted as used every time it is used as a configuration for the compiler, that is, every time it is used to compile a program. On the other hand, a single heuristic is counted as used every time it is evaluated, that is every time the actual values of the features are substituted into the variables appearing in the formula of the heuristic. This means the heuristic is counted as used every time a decision is made based on its value. For example, a heuristic determining the unrolling factor of loops will be evaluated (and therefore used) every time an unrollable loop is being compiled.

The scores of both the heuristic sets and the heuristics are updated as follows:

$$score_{updated} = \frac{score_{old} * usecount_{old} + score_{new} * validusecount_{new}}{usecount_{old} + usecount_{new}} \tag{3.2}$$

where $score_{old}$ and $usecount_{old}$ are the score and use count stored in the knowledge base, $score_{new}$ is the speedup measured during the test execution, $usecount_{new}$ is the number of times the heuristic has been used during the test execution. If we are updating the score of a heuristic set, $usecount_{new}$ will always be equal to one. $validusecount_{new}$ is the number of times the result of the evaluation of the heuristic was inside the validity range:

$$validusecount_{new} = usecount_{new} - (tooLow + tooHigh)$$

with $tooLow$ (respectively $tooHigh$) being the number of times the heuristic was evaluated lower (resp. higher) than min (resp. max).

Immediately later, the use count is incremented too:

$$usecount_{updated} = usecount_{old} + usecount_{new}$$

On the other hand, it might happen that a candidate configuration fails to generate a working executable, usually because some optimizations of the underlying compiler cannot be applied at the same time. If a configuration failed, we want to penalize it. Therefore we use a $score_{new}$ equal to zero and a $usecount_{new}$ equal to 1, thus decreasing its score, especially if this happens when the heuristic that led the configuration to fail is fairly new and its $usecount_{old}$ is low.

As it can be seen from Equation (3.2), the score that is computed along the life of the knowledge base is the expected speedup, of each heuristic set and of each heuristic. It is used by the algorithm to determine the best known candidates: the higher the score, the better the expected speedup while using the heuristic.

Rationale The current scoring system was chosen because it proved experimentally[3] to be fast at learning when no knowledge is available, while keeping the ability to improve later on. Furthermore, the method we use to update the score of each heuristic scales well with the number

[3]This experimental choice does not reduce the generality of our approach, because only a reduced number of programs (namely `bitcount`, `qsort1` and `susan_c`) where used during the development of the GCC version to make this decision. Later, performance results over the full set of benchmarks and the implementation in a second compiler just confirmed the quality of the chosen scoring system.

of their executions (the cost of update is constant). This is extremely important, since we collect data for every single candidate we test.

It is worth noting that we considered multiple scoring algorithms for long-term learning. At first, we considered an algorithm that sorted the candidates by speedup, then assigned a score based on the ranking (for n candidates, decreasing from n points for the best one to 1 for the worst one). Such an algorithm is more robust to measurement disturbance during the test phase, since the score is unchanged as long as there is no switch in the relative positions of one candidate with respect to another one, whereas using the speedup is sensitive to such errors. We decided to use the speedup itself, instead of the speedup ranking, to determine the score for two reasons. First, it provides useful information discerning how better a candidate is compared to another one (*e.g.* being twice as fast is not as good as being four times as fast). Second, it takes into account whether a candidate provides an actual improvement: using the ranking-based system, if all the candidates in a set are slower than the default configuration, the best one would still get the maximum number of points, making it an apparently good candidate. This problem could be partially solved by assigning zero points to candidates with speedup below 1, but this would still lose interesting information: a candidate scoring just below 1 might be a good one when applied to a different program, and penalizing it too is likely a short-sighted decision.

3.1.7 Compiler performance over time

The long-term learning algorithm does not have an initial training phase: it can immediately be used. Therefore it is important to determine what is the performance of a pristine system based upon such algorithm.

A strict requirement of the algorithm is the availability of a default configuration, to be used as the comparison term to compute the speedup of each candidate. It can be a configuration where all the optimizations are disabled and all the numeric parameters of optimization algorithms use generic values: the algorithm does not require any pre-existing knowledge about the underlying hardware architecture and the expected benefits of the compiler optimizations. It is its task to figure this out. Nevertheless, if there is already available knowledge, the algorithm is able to exploit it. The best way to do this is to use such knowledge to define a non-trivial default configuration.

Since the default configuration is used as the comparison term, only the candidates improving upon it will receive a good score: the default configuration can be seen as the worst-case fallback the system uses when it cannot find a better configuration.

In the limit, for an infinite number of compiled programs, the system tends to converge to the optimum, whatever the starting point.

The ideal usage for long-term learning would be to enhance the performance portability enabled by iterative compilation algorithms [34]: the compiler could be distributed with a generic configuration, containing a set of heuristics good for a family of architectures (*e.g.* x86-i386 architectures). While a program is being developed, every time it is compiled, the compiler evolves as well, adapting itself to the specific system it runs on. When it is time to release the production binary, the evolution of heuristics can be disabled if the need to keep the performance predictable arise. In order to further speedup the evolution of heuristics, the knowledge base can be shared by all the developers working on a project. This would also prevent them from dealing with different binaries.

Furthermore, if multiple candidates known to be good are available, the system could implicitly race them to find the best one: instead of starting from an empty knowledge base, it could be initialized with the set of candidates to be raced, all with the exact same score. The system would thus start from such heuristics before using randomly generated ones, and would exploit them if they proved to provide speedups.

The IID model presented in [2, 45] aims at being an online probabilistic tuner, so it is in some way similar to long-term learning regarding the considerations proposed in this Section. Still, there are some important differences. First of all the fact that it needs some offline tuning to be done (the issue was later solved in [47]). Moreover, it does not take into consideration the fact that applying different code transformations at the same time can have effects that are not independent from one another. Long-term learning, on the other hand, learns heuristic sets as a whole, thus solving this issue.

Long-term learning is meant to constantly improve the performance of the compiler over its whole life. It would be unrealistic to think that the target system is never going to change and that the compiler itself is never going to be updated. Therefore, the algorithm was designed from the beginning with the objective of being robust to this kind of events.

This paragraph presents how this is possible in different scenarios.

The target system changes First of all, let us consider the case of the unmodified compiler with a target system that changes (a new processor is installed, the amount of available RAM is increased, a new version of the operating system is installed, *etc.*). If the current best heuristics sets performs well with the new system configuration, nothing will change.

On the other hand, if some new heuristic obtains better performance, it will be chosen more often for originating new candidates. At the same time, the score of the old heuristics will slowly decrease, until they drift into oblivion.

Having a new heuristic reach the top of the ranking might seem hard, but it is not. New heuristics are able to overcome the old ones because all the scores are determined by the speed-ups weighted by the number of times every heuristic has been used. Therefore, old heuristics are resilient to changing their score (since they have proven their average speedup over time) but new ones have low weight and their score can immediately obtain a really high value if they provide a high speed-up.

Changing program features availability Let us consider now the availability of a new version of the compiler, able to compute a new program feature. All the old heuristic formulas are still valid. During the evolution and mutation processes, new formulas are generated, and some of them will use the new feature. If they manage to get higher speed-ups, they will earn higher scores and will be used more frequently.

In the opposite scenario, should the compiler lose the ability to compute a feature, the heuristic formulas containing that feature will not be able to be evaluated any more. This means their score will drop fast, and they will stop being used.

New heuristic needed Let us consider the scenario where an updated version of the compiler has to take a new decision and therefore needs a completely new heuristic. Every candidate configuration used in the past, is composed by a set of heuristics (one for each decision point in the compiler). The names of all the needed heuristics are stored in set N of Algorithm 4.2.1. When the compiler requires a new heuristic, its name is added to N. The GENERATEMISSINGHEURISTICS function is called immediately before using each candidate. If a name in set N has not a corresponding heuristic inside the candidate, GENERATEMISSINGHEURISTICS will generate the formula for such heuristic, completing the candidate.

One compiler, different histories With long-term learning, every deployed compiler has its own compilation history and has generated different heuristics, thus behaving differently from every other compiler. This might have downsides, especially in a production environment, when it comes to bug reporting. On the other hand, it should be considered that this is no different from every other machine-learning based compilation approach: after the training phase, each of these compilers has its own, unique, model, and modifies its own behavior accordingly. We designed the knowledge base to be as small as possible and contained in a single file. This allows different developers to easily exchange their history, when needed, such as for bug fixing or for comparing performances. Furthermore, the learning process can easily be disabled, when needed, using only the current best heuristics. This allows to have a traditional compiler, but with ad-hoc heuristics, once the desired performance level has been achieved. As for other machine-learning approaches, deciding when to stop the learning process is up to the user: the longer, the better. But unique to long-term learning is the possibility to never stop it: idle CPU times could be used to compile more programs, therefore further improving the knowledge and the performance.

3.2 Experimental setup

In order to show the viability of our approach and its applicability to multiple compilers, we implemented long-term learning into two different compilation toolchains (GCC and PetaBricks), and performed tests on multiple hardware configurations.

In the GCC implementation the heuristics are used by a script to toggle GCC command line flags. In the PetaBricks implementation the heuristics are evaluated directly by the compiler, both to define parameters used by PetaBricks optimizations and to define parameters passed to the underlying GCC.

The long-term learning algorithm implementation is divided into two parts. The learning algorithm itself is implemented in Python [125] as a framework, and is common to all the compilers. It chooses and learns the heuristics. The second part is implementation specific and it gives the compiler a way to provide the program features to the algorithm and to use the chosen heuristics to make decisions about the compilation process. The compiler-specific implementations will be described in Section 3.2.1 and 3.2.2. In these experiments, most of the times the heuristics are then used to decide the best command line parameters, but this is just one possible use: in fact, the heuristic for the Petabricks tiling algorithm is evaluated directly inside the compiler itself.

As it can be seen in the *Configuration* section of Algorithm 4.2.1, long-term learning needs a few parameters to be set in order to define its behavior. We determined the values to assign to such parameters (in Table 3.1) through a brief experimentation.

Our implementations were tested on different hardware configurations, to show that the algorithm is able to adapt compilers to multiple architectures.

Hardware Configuration 1 A machine with 4 Intel Xeon X7550 processors (8 physical cores each) running at 2.00 GHz with 128 GB of RAM.

Hardware Configuration 2 One core of a machine part of the CILEA Lagrange cluster [7]. Each machine is equipped with two Intel Xeon X5460 processors (4 physical core each) running at 3.16 GHz and 16 GB of RAM.

Hardware Configuration 3 One processor of a machine equipped with two Intel Xeon X5460 processors (4 physical cores each) running at 3.16 GHz and 8 GB of RAM.

We want to show that the knowledge acquired by the compiler through long-term learning can be applied to improve the performance of new programs. Therefore, we need to simulate the life of a compiler that has compiled a certain number of programs previously, showing how this affects the performance of an unseen program.

We have a set of available programs, taken from a benchmark suite. For every simulation, we use one of the programs of the suite as the test program and the rest as the training-set, in a Leave-One-Out Cross-Validation [72] configuration.

Each simulation is performed as follows. First of all, the test program is compiled using the default compiler configuration to find out the reference speed. Then, a program from the training set is compiled. Then, the test program is compiled again, using the knowledge acquired by the compilation of the first training program, and is then tested recording its execution time. Later, we compile the second program from the training set and then again we compile and run the test program, using the knowledge acquired from the compilation of the first two training programs. This procedure is repeated for each program in the training set.

It should be noted that, in order for the experiment to be meaningful, it is mandatory that the test program has never been seen by the system before, to prevent the compiler from acquiring knowledge from its compilation. In the simulation setup that was just described, the test program is compiled multiple times. In order to allow this, a special option was implemented in the compilers, enabling a program to be compiled using the current knowledge, but disabling write access to the knowledge base. Such option is used every time the test program is compiled, therefore, every time, the system behaves as if it never saw such program before.

It is also worth noting that we use the term ”training set”, but this does not mean that a training phase is needed: it just denotes the set of previously compiled programs, that provide the data for the learning process. In a production system, such programs would be the programs compiled by the compiler during its lifetime, up to the current one. The compiler is immediately usable, without an explicit initial training phase.

Since the long-term learning algorithm performs many operations (like generating formulas) randomly, it is expected that two different simulations will yield different results. Therefore, in order to provide more statistically relevant information, all the graphs shown in Section 3.3 are plotted using data computed as the average execution time computed over 5 runs.

The next two sections will describe compiler-specific details of our implementations.

3.2.1 GCC

We implemented long-term learning on top of GCC 4.6.3. The compiler specific part consists of two components.

The first component is a GCC plugin, activated during the compilation process to extract the static features (that is, extracted from code analysis, as opposed to dynamic features, extracted from runtime information) of the program. The computed values refer to the GIMPLE [89] intermediate representation of GCC. The choice of what features to use is important, because only a set of features providing actual information about the program allows to build useful heuristics. So, we decided to use a subset of the features proved valid by [94], namely those listed in Figure 3.5, each averaged on the values computed for the whole program being compiled. In particular, the features we chose are simple but able to describe the structure of the program: they allow us to automatically find out heuristics at least as good as the hand-written ones provided by GCC. We might implement more feature extractors in the future, to better characterize the programs in order to further optimize them. The second component is the interface between the Python learning framework and GCC itself. It is responsible for evaluating the chosen heuristics and invoking GCC itself enabling the optimizations through GCC command line flags according to the evaluated heuristics. Unfortunately,

Number of basic blocks in the method
Number of basic blocks with a single successor
Number of basic blocks with two successors
Number of basic blocks with more than two successors
Number of basic blocks with a single predecessor
Number of basic blocks with two predecessors
Number of basic blocks with more than two predecessors
Number of basic blocks with a single predecessor and single successor
Number of basic blocks with a single predecessor and two successors
Number of basic blocks with two predecessors and a single successor
Number of basic blocks with two predecessors and two successors
Number of basic blocks with more than two predecessors and more than two successors
Number of basic blocks with a number of instructions less than 15
Number of basic blocks with a number of instructions in the interval [15, 500]
Number of basic blocks with a number of instructions greater than 500
Number of edges in the control flow graph
Number of assignment instructions in the method
Number of conditional branches in the method
Number of unconditional branches in the method
Number of binary interger operations in the method

Figure 3.5: The list of features used in heuristic formulas.

GCC was not built as a research compiler and it doesn't allow full control over the specific optimizations to be enabled, since many of them depend on a specific optimization level being activated. Therefore, instead of just learning the list of optimization passes to enable, long-term learning actually learns both the optimization level (between `-O0` and `-O3`) and the list of optimization flags to enable (`-f<optimization_name>`) or disable (`-fno-<optimization_name>`) explicitly. Is is known that some of GCC's optimizations are disabled by default because they are unsafe. We deal with this by automatically discarding executable that turn out to be not working after being generated.

The benchmark suite we used for GCC is cBench [40]. It is composed by sequential C programs (listed in Figure 3.6) and derives from MiBench [52], with modifications aimed at making it better to test the efficiency of compiler optimizations. We chose this benchmark suite instead of other well known ones (such as SPEC) because it is used by many other works dealing with machine learning and iterative compilation, making it easier to compare the results. In cBench, each benchmark is associated with a number representing how many executions (usually hundreds) of the program are required to obtain a total running time of about 20 seconds for the unoptimized version. We use such total running time as the value we compute the speedup on, because the multiple

automotive_bitcount	consumer_tiff2bw	security_pgp_d
automotive_qsort1	consumer_tiff2rgba	security_pgp_e
automotive_susan_c	consumer_tiffdither	security_rijndael_d
automotive_susan_e	consumer_tiffmedian	security_rijndael_e
automotive_susan_s	network_dijkstra	security_sha
bzip2d	network_patricia	telecom_adpcm_c
bzip2e	office_ispell	telecom_adpcm_d
consumer_jpeg_c	office_rsynth	telecom_CRC32
consumer_jpeg_d	office_stringsearch1	telecom_gsm
consumer_lame	security_blowfish_d	
consumer_mad	security_blowfish_e	

Figure 3.6: The list of programs used for the experimental tests.

executions smooth out measurement errors. cBench contains multiple datasets for each program. Nevertheless, for most experiments we just used one dataset for each program, because [41] showed that it sufficient to achieve an optimization level within 5% of the best possible level. The experiments presented in Section 3.3.1 show what happens when multiple datasets are used.

cBench programs are sequential, so they are not influenced by the fact that we are using parallel hardware to run them.

3.2.2 PetaBricks

PetaBricks [5] is an open source compiler and programming language developed at MIT that uses machine learning and evolutionary algorithms to autotune [6] programs, by making both fine-grained and algorithmic choices. PetaBricks programs work on numerical matrices as their input and output data. The compiler produces executables that expose hooks allowing the autotuner to adapt them to the underlying platform and to a given input data size.

PetaBricks does not compile C programs, therefore, we used its own benchmark suite instead of cBench to perform our tests.

The PetaBricks compiler is a source to source translator, reading its own input format and generating C++ programs that will be later compiled by GCC, used as the backend compiler. Therefore, we reuse part of the setup from the GCC long-term learning compiler. In particular, we use the same GCC plugin to compute features about the program being compiled. The plugin works at GIMPLE level, so the source language (C for the cBench tests, C++ for PetaBricks-generated code) is not important. At first, we considered using features collected from PetaBricks's own intermediate representation (IR), but PetaBricks has not many optimizers requiring heuristic decisions, so we mainly use them to direct the optimizers of the underlying GCC. Therefore, such IR is too high level to be useful for our actual setup. Though, given more PetaBricks optimizers, PetaBricks' own IR would provide more interesting features.

The interface between the long-term learning framework and PetaBricks is written in Python, as for GCC, but instead of evaluating the heuristics it just stores them in a configuration file passed to PetaBricks as a parameter. We modified the PetaBricks compiler adding a component, written in C++, able to evaluate the heuristics. This shows that it is possible to use long-term learning with different levels of integration with the compiler it is working on.

The optimizations the PetaBricks implementation works on include a PetaBricks-specific parameter, influencing the code it generates: the number of times matrices have to be split by the tiling optimizer of PetaBricks. Furthermore, we learn the numerical optimization parameters that are passed to the underlying GCC compiler, namely those related to function inlining (`max-inline-insn-auto`, `max-inline-insns--single`, `large-function-insns`, `large-function-growth`, `large-unit--insns`, `inline-unit-growth`) and to loop unrolling (`max-unroll-times`, `max-unrolled-insns`), because PetaBricks generates C++ code containing many small functions and many loops. We also learn all the flags described in Section 3.2.1 for the GCC implementation.

We learn more optimization parameters than in the GCC implementation: this is not related to the speed of the learning process or the speedups we obtain with or without them. It is only meant to show the flexibility of our algorithm. In the GCC implementation we learn heuristics to make binary decisions about which optimizations to enable. Here, we also show the possibility to learn heuristics that evaluate to numerical results, both for direct use with PetaBricks optimizations or with optimization of underlying GCC.

We choose not to integrate the long-term learning algorithm and the autotuner because autotuning aims not only at improving the performance of the program, but also at making it able to adapt to changing hardware without the need to recompile the software [5]. Long-term learning only modifies the program at compile-time. On the other hand, as a future work, long-term learning could be used to automatically determine good default values for all the parameters set by the autotuner, removing the need to autotune the program and enabling immediate execution as long as the actual runtime system is the same as the target system considered at compile-time.

3.3 Experimental results

In this section we will present the results of the tests we performed using the described experimental setup, first for GCC, then for PetaBricks. Unfortunately, it is not possible to provide a direct comparison with other learning-based systems, since the main aim of long-term learning is to be able to find out heuristics for compilers that have none, while other systems aim to improve the performance of the existing heuristics.

On the other hand, it is possible to show the characterizing feature of our system, that is, the way it gradually evolves the heuristics to find the best performing ones. This will be done in the last subsection of the experimental results.

3.3.1 GCC

We performed multiple tests using GCC. We used both `-O0` and `-O3` as the default configurations. The first configuration, meaning no active optimizations, allows us to verify the ability of long-term learning to automatically find out good heuristics for a compiler with not even a default set of heuristics defined yet (here, we aim at reaching the same performance as `-O3` from `-O0`). The second configuration enables most of GCC optimizations, generates very fast programs and is used as a reference by most iterative compilation works. When using it as the default configuration, we aim at learning heuristics able to improve over its performance.

Figure 3.7 shows the results of long-term learning on Hardware Configuration 1, using `automotive_susan_c` as the test program. The initial, unoptimized, execution time drops rapidly as soon as knowledge from the compilation of at least one other programs has been acquired, because the system manages to find out a set of heuristics enabling the optimizations that deliver the most speedup. Later, compiling more programs, the execution time continues to decrease slowly, as marginally better heuristics are found.

The number of different configurations that are tested during one compilation with long-term learning is determined by parameter k of Algorithm 4.2.1. Figure 3.8 shows that, on Hardware Configuration 1, using just 5 candidates per generation leads to a slowly declining learning curve, whereas using more than 6 candidates per generation does not provide significantly faster learning, therefore we chose to set the default to 6. We also determined experimentally that our algorithm performs well with just 3 generations. Therefore, with less than 20 candidate configurations tested during each compilation we can obtain

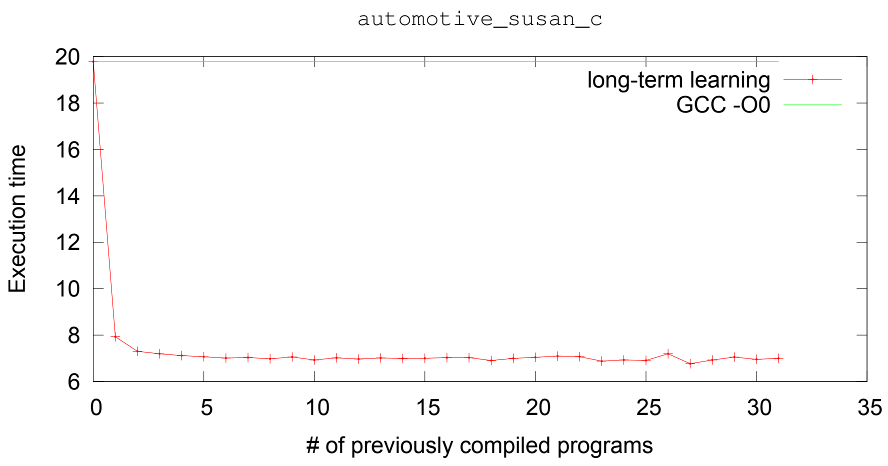


Figure 3.7: Execution times of `automotive_susan_c` on Hardware Configuration 1 after the compilation of an increasing number of other unrelated programs. The default configuration is `-O0`.

good results without the need for an initial training phase, whereas other iterative approaches need hundreds of them. It should be noted that using 6 candidates over 3 generations leads to better results than just compiling 18 candidates in a single generation. This is due to the fact that using a single generation all the candidates are randomly generated starting from the pre-existing knowledge of the compiler, whereas using multiple generations the candidates of each generation are based upon the knowledge acquired during the previous one for the specific program being compiled.

Figure 3.9 shows the execution times for the same test, `automotive_susan_c`, on Hardware Configuration 2, using different training sets. Training set 1 is the same used for Figure 3.7. The training sets 2 and 3 include the same programs of training set 1, but in a different order. As it can be seen, the behavior of the different training sets is comparable. The descending trend is analogous for the three sets. Training set 2 is included in the graph to show that even if the initial choice of heuristics is not optimal, the performance tend to improve and to converge to the minimum over time. Therefore, except for the very first programs, the specific sequence of previously compiled programs is not much important in determining how well the next programs will be compiled.

Figure 3.9 also contains another important information. It shows us that long-term learning is able to devise heuristics that not only improve the performance of the default configuration, but reach and surpass the

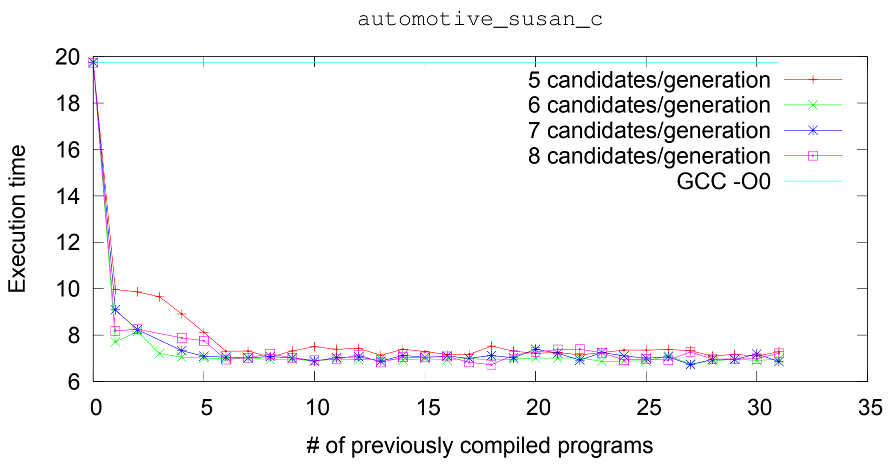


Figure 3.8: Execution times of `automotive_susan_c` on Hardware Configuration 1 for varying number of candidates used for each generation. The default configuration is `-O0`.

performance of `-O3`. Unfortunately, though, this does not always happen: Figure 3.10 shows an example of a test program where such that, on average, long-term learning is able to optimize with respect to the provided default configuration, but without reaching the performance of `-O3`. Nevertheless, given enough previously compiled programs, or a specific execution where the long term learning algorithm explores the heuristics search space in a particularly convenient way, it will eventually converge to the best possible solution, improving over `-O3` where possible.

If the aim is to actually improve the performance of a higher optimization level, such as `-O3`, the best approach is to set such level as the default configuration to compute the speedup against. This way, as described in Section 3.1.6, all the configurations yielding an execution time higher than the default configuration will be penalized: the algorithm selects heuristics actually improving over the default one, if they exist.

Figure 3.11 provides an example of such an improvement, recorded on Hardware Configuration 1. It shows how long-term learning is able to improve the performance of the `telecom_gsm` benchmark over GCC `-O3`. It is worth noting that the improvement is not as stable as those over `-O0`. This is due to the fact that the execution times of good configurations are close to those of bad configurations. Therefore, the score assigned to good heuristics sets is only slightly higher than that of bad sets, making the choice of the best ones more tricky.

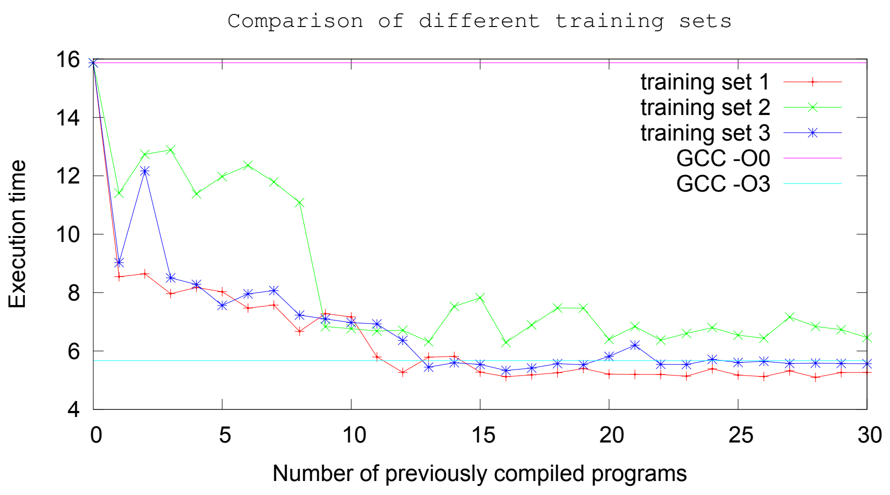


Figure 3.9: Execution times of `automotive_susan_c` on Hardware Configuration 2 after the compilation of an increasing number of other unrelated programs. The test is repeated using different training sets. Execution times improve both on `gcc -O0` and on `gcc -O3` for training sets 1 and 3. The default configuration is `-O0`.

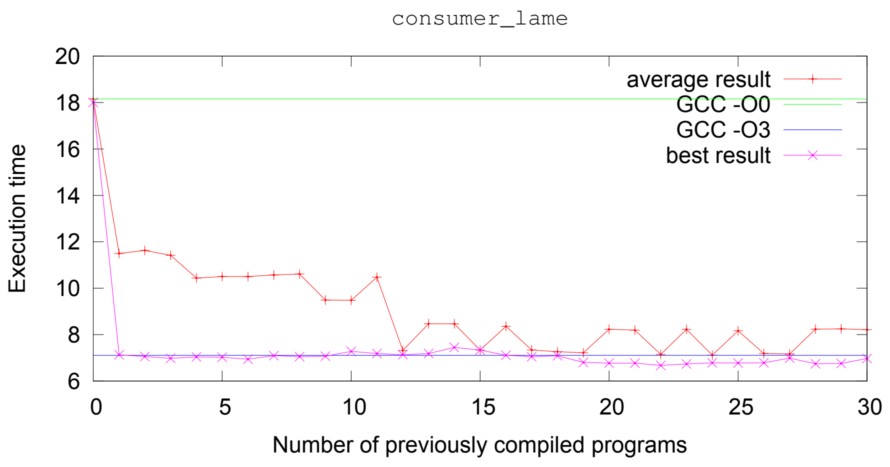


Figure 3.10: Execution times of `consumer_lame` on Hardware Configuration 2 after the compilation of an increasing number of other unrelated programs. The average execution times improve on `gcc -O0` but not on `gcc -O3`. Though, they can be surpassed if the random generation of heuristics explore the right area of the candidate heuristics search space.

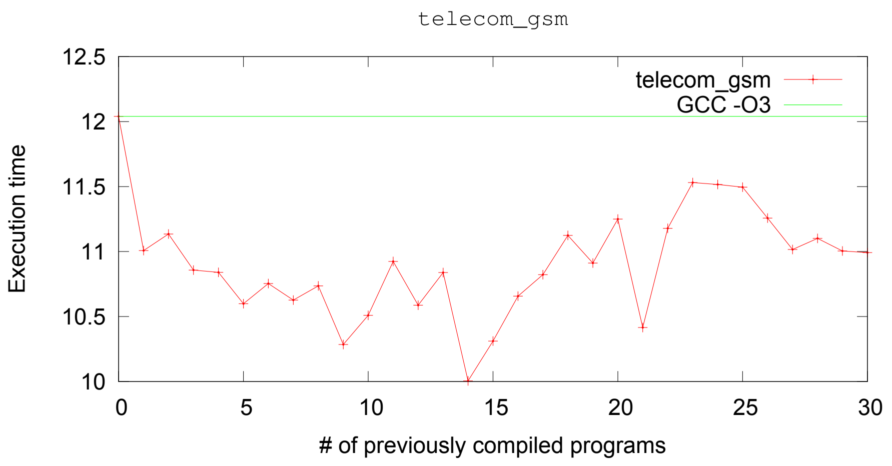


Figure 3.11: Execution times of `telecom_gsm` on Hardware Configuration 1 after the compilation of an increasing number of other unrelated programs.

If the aim is to actually improve the performance of a higher optimization level, such as `-O3`, the best approach is to set such level as the default configuration to compute the speedup against. This way, as described in Section 3.1.6, all the configurations yielding an execution time higher than the default configurations will be penalized: the algorithm selects heuristics actually improving over the default one, if they exist.

Figure 3.11 provides an example of such an improvement, recorded on Hardware Configuration 1. It shows how long-term learning is able to improve the performance of the `telecom_gsm` benchmark over GCC `-O3`. It is worth noting that the improvement is not as stable as those over `-O0`. This is due to the fact that the execution times of good configurations are close to those of bad configurations. Therefore, the score assigned to good heuristic sets is only slightly higher than that of bad sets, making the choice of the best ones more tricky.

It is interesting to inspect the behavior of long-term learning over a longer time span (90 programs in the training set instead of 30), shown in Figure 3.12. Here we can clearly identify three different phases. At the beginning (until 10 programs have been compiled) the learning process is really fast, since it is easy to learn optimizations that provide a huge speedup. Actually, some overfitting takes place: the first 10 programs of this test contain some similarities with `automotive_susan_c`, therefore its performance improves very fast. Nevertheless, the aim of long-term learning is to optimize the compiler to perform better on average, for all

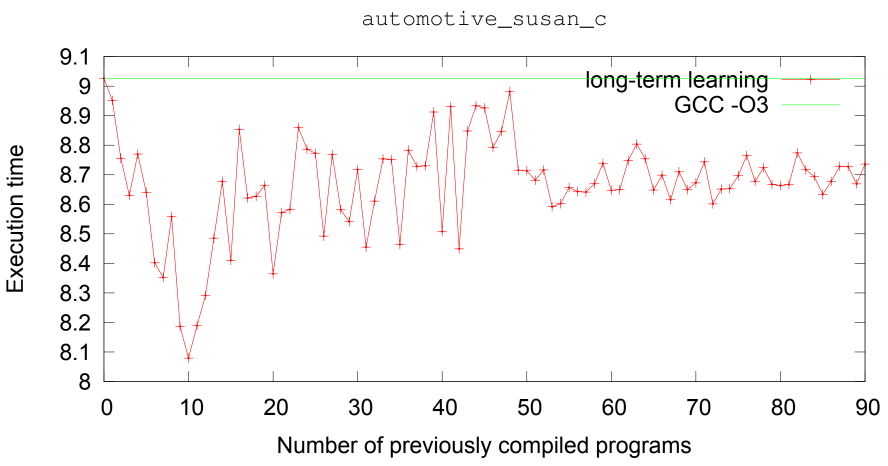


Figure 3.12: Execution times of `automotive_susan_c` on Hardware Configuration 1 after compiling an increasing number of other unrelated programs. The performance level improves while becoming more stable over time.

programs. The test program is not treated in a special way. Therefore, as the long-term learning-enabled GCC goes on compiling new programs from the training set (programs between 10 and 50), the performance of `automotive_susan_c` get worse, but are unstable. After the first 50 training programs have been compiled, enough knowledge has been gathered for the performance to become more stable and predictable. It is likely that, given even more training programs, after finding out an heuristic set that is good, on average, for all programs, the system would tend to converge to a heuristic set that is able to predict the best configuration for every program, therefore further improving the performance again.

Figure 3.13 shows the average and maximum speedup obtained by each of the test programs along the lifetime of the compiler over the default configuration `-O0` on Hardware Configuration 2, and compares them with the speedup over GCC `-O3`. More precisely, let s_i be the speedup of the test program after compiling i training programs, and let n be the total number of training programs. The values plotted in the bar chart are computed as:

$$\text{average speedup} = \frac{\sum_1^n s_i}{n}$$

and

$$\text{maximum speedup} = \max(\{s_1, \ldots, s_n\})$$

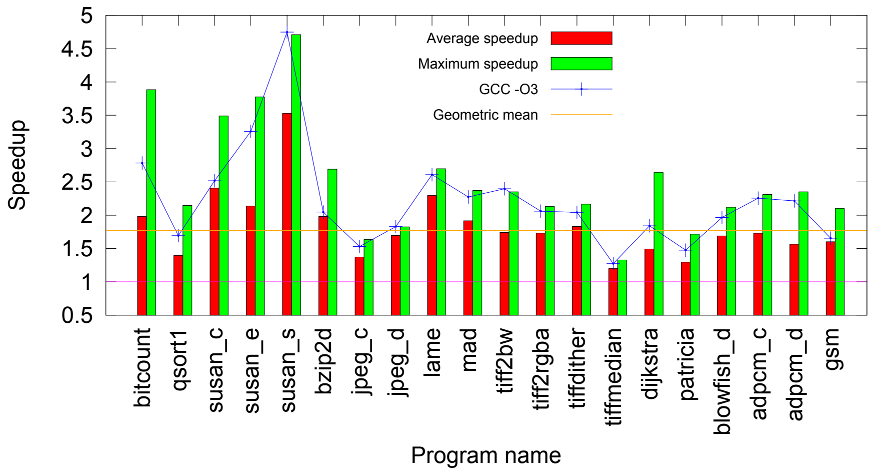


Figure 3.13: Speedup obtained by various programs with respect to `gcc -O0` on Hardware Configuration 2, and compared to `gcc -O3`. The average and maximum speedup are computed over all the values obtained by testing the program once after compiling each of the programs listed in Figure 3.6.

The average speedup includes the small speedups obtained after compiling the first few programs, so, obviously, it is worse than GCC `-O3`. But in the end, as by our aim, the algorithm is able to find heuristics good enough to reach and surpass `-O3`, as shown by the maximum speedup.

Figure 3.14 shows the average and maximum speedups obtained over GCC `-O3`. As it can be seen, the speedups obtained over the default configuration `-O3` are not too big, but still comparable to those of GCC profile driven optimization. This was expected. The main aim of the long-term learning algorithm is to find a model of the target architecture, in the form of a good compilation heuristics set, without the need for human intervention in the process. The data we gathered show that this actually happens: the algorithm is able to reach the performance of GCC's maximum optimization level after the compilation of just a short number of programs, without the need for any training phase. Ongoing work is trying to always ensure a sensible improvement over `-O3`. The main obstacle preventing improvements over `-O3` is the small speedup each of them can provide, sometimes comparable with the measurement errors and therefore shielded by this. Therefore, such improvements are difficult to detect and it is not easy to select the best heuristic.

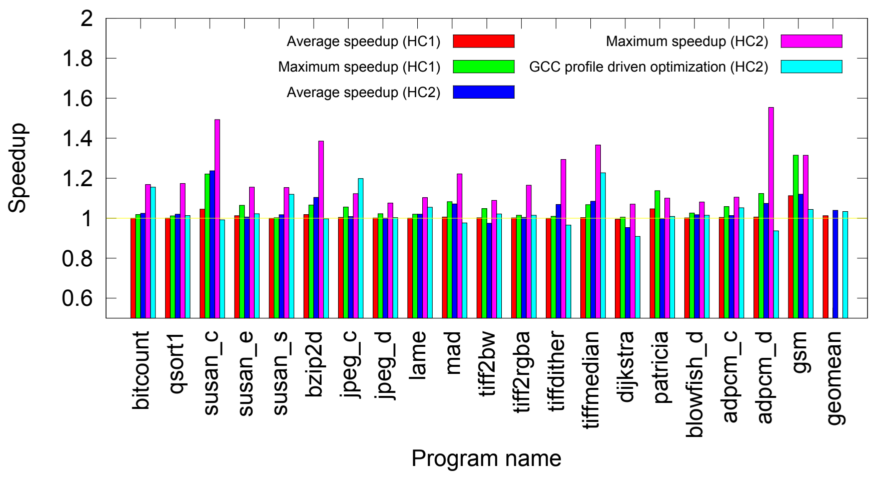


Figure 3.14: Speedup obtained by various programs with respect to `gcc -O3` on Hardware Configuration 1 (HC1) and 2 (HC2). We represent the average and maximum speedup computed over all the values obtained by testing the program once after compiling with long-term learning each of the programs listed in Figure 3.6. The speedups provided by GCC profile driven optimization are presented as a comparison. The last bar, *geomean* is the geometric mean of the measured speedups, computed for HC1, HC2 and HC2 with profile driven optimization.

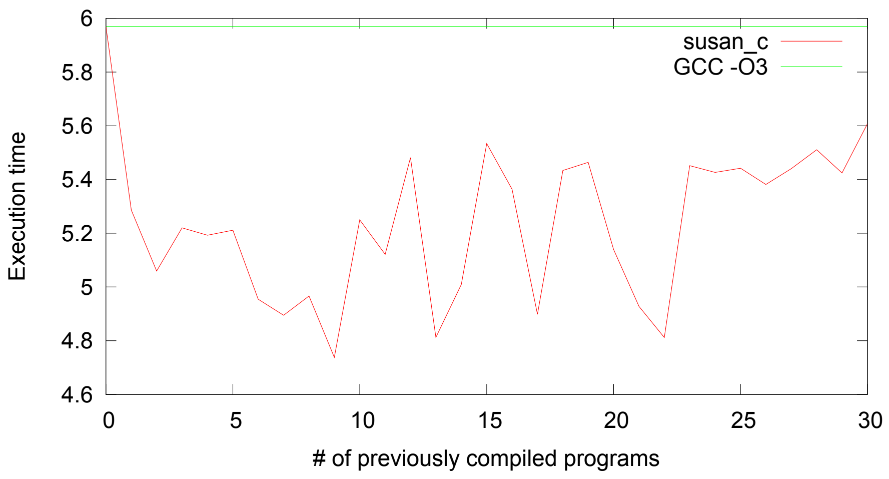


Figure 3.15: Execution times of `automotive_susan_c` on Hardware Configuration 2 after the compilation of an increasing number of other unrelated programs. The default configuration is `-O3`. For each generation of candidates, a different dataset is used.

Experiments with multiple datasets

For the sake of completeness, this section presents the results of some experiments where multiple datasets have been used while testing GCC programs, instead of just one. To be precise, the existence of multiple generations of candidates during the long-term learning compilation process is exploited. When the candidates of each generation are tested, they are all run using the same dataset, therefore allowing for an easy comparison of the running time results. But, every time a new generation of candidates is created, it is tested using a different dataset. With the default settings of long-term learning, each compilation comprises three generations: this allows the resulting program to be generated by heuristics evolved over at least three datasets, just for the current program.

On top of this, it should be considered that the heuristics are not evolved from scratch for the single program, but they are the result of the long-term learning over multiple programs. Therefore, it should not be surprising that looking at the evolution of heuristics as depicted in Figure 3.15 and Figure 3.16, there is no apparent difference with the corresponding figures built using a single dataset: if a heuristic is robust enough to work on multiple programs, it is even more so when just the dataset is changed, keeping the same program. In fact, the algorithm

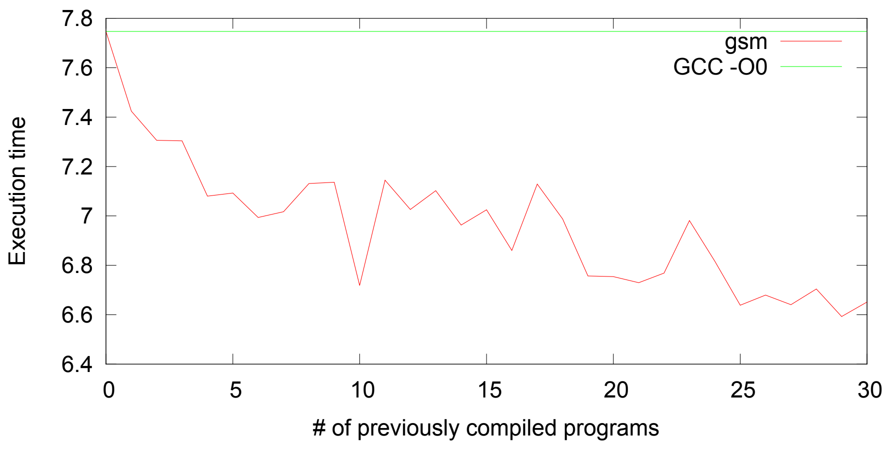


Figure 3.16: Execution times of `automotive_susan_c` on Hardware Configuration 2 after the compilation of an increasing number of other unrelated programs. The default configuration is `-O3`. For each generation of candidates, a different dataset is used.

is able to improve the performance over the `-O3` optimization level of GCC.

So, this shows that the reason why multiple datasets have not been used for all the tests is just that they do not convey any sensible additional improvement to the performance of the resulting heuristics.

3.3.2 PetaBricks

The amount of data we gathered about PetaBricks is smaller than that available for GCC, because the running time of the tests is longer, for various reasons. First of all, the number of tests contained in PetaBricks's benchmark suite is bigger than that of cBench. Furthermore, every program compiled by PetaBricks needs to be tuned by the underlying autotuner that adapts runtime parameters exposed by each program to the system it is running on, using the algorithm described in [6]. It is a genetic algorithm, and as such it requires the program to be run multiple times to find out the best configuration. This implies that every program version generated by our long-term learning algorithm has to be tuned before being run to measure its execution time and determine its speedup. Since the tuning process requires at least some minutes, the added delay with respect to the GCC implementation is not negligible.

PetaBricks programs are natively parallel, so they tend to occupy all the resources of the machine. In order to speedup the execution of

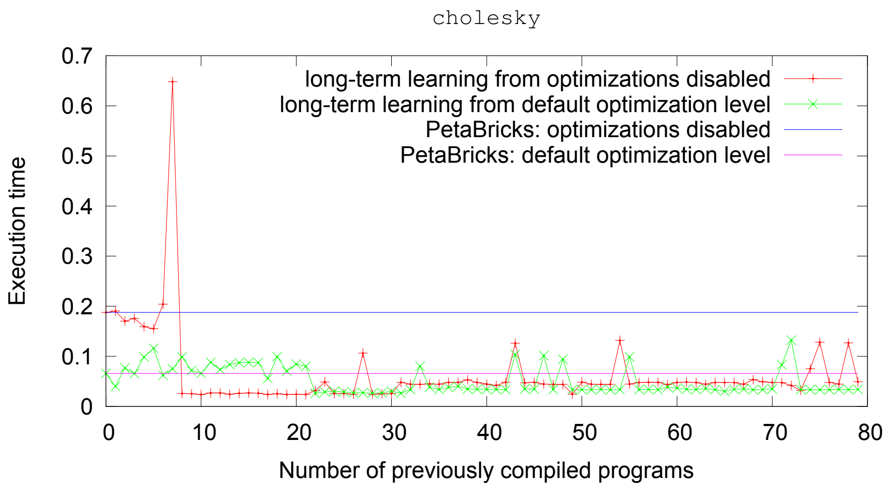


Figure 3.17: Execution times of `cholesky` on Hardware Configuration 1 after the compilation of an increasing number of other programs. The speedup is computed w.r.t. PetaBricks with no optimization knowledge at all (optimizations disabled) and from the default optimization level.

the tests, we used multiple machines and we executed multiple tests in parallel using MapReduce according to the approach described in Chapter 4 and in [119].

Even if the time required to simulate lifelong evolution of PetaBricks with long-term learning is long, the results are promising.

We show the running times of a test computing the Cholesky decomposition of a square matrix of size 256 (Figure 3.17). The default compiler configurations used as reference for the long-term learning algorithm are: optimizations disabled (activated by the `-O0` parameter in our modified PetaBricks version) and default (that is, maximum) PetaBricks optimization level (does not require any particular command line option).

The spikes appearing in the graph are due to the underlying autotuner. Being based upon a genetic algorithm, it is not assured to always converge to the same solution, especially when the executable it is run on is a different version of the same program. Therefore, it is possible that some configurations generated by long-term learning turn out to be harder to tune than most others. We can see that after an initial slow descent phase, long term learning is able to find a good configuration. Moreover, the performance level reached is better than that of the default optimization level of PetaBricks.

The same happens in Figure 3.18, when using a square matrix of size 1024.

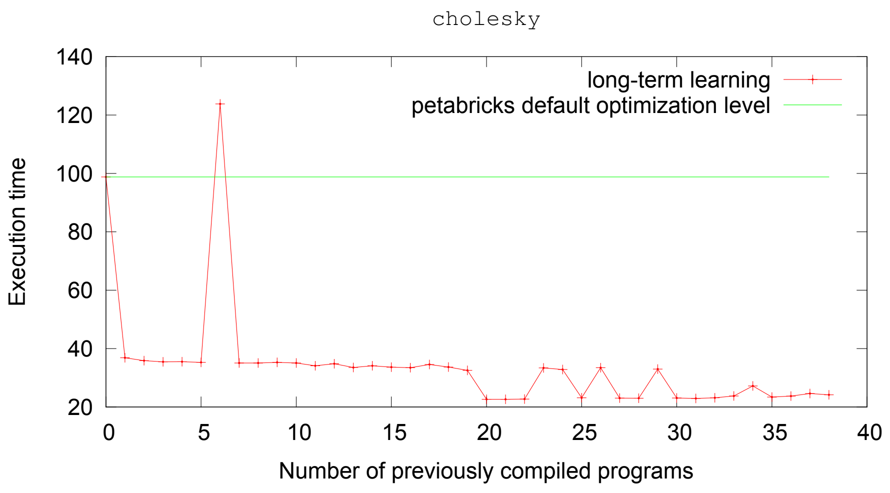


Figure 3.18: Execution times of `cholesky` on Hardware Configuration 1 after the compilation of an increasing number of other programs. The speedup is computed with respect to PetaBricks default optimization level.

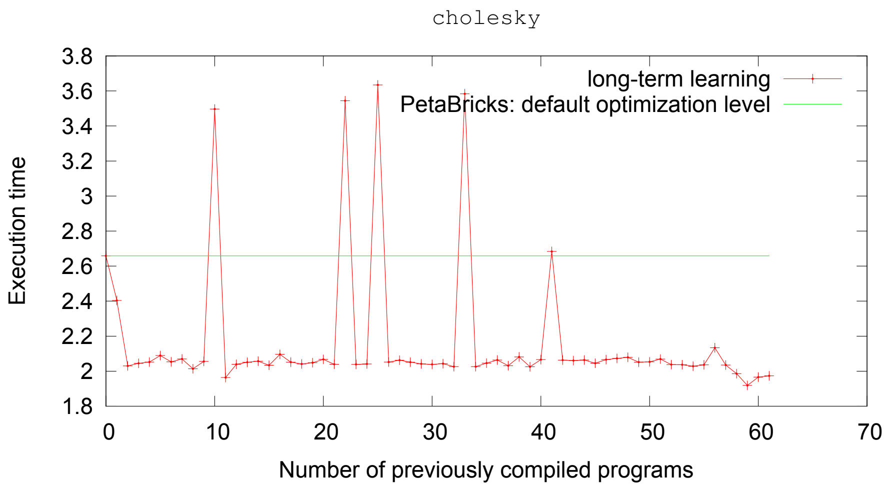


Figure 3.19: Execution times of `cholesky` on Hardware Configuration 3 after the compilation of an increasing number of other programs. The speedup is computed w.r.t. PetaBricks default optimization level.

Figure 3.19 shows that a comparable reduction of the running times can be obtained on a different hardware configuration too (namely, Hardware Configuration 3).

3.3.3 Evolution of Heuristics

This section will show examples of how heuristics are created and evolved by the long-term learning algorithm along the lifetime of a compiler.

Showing on paper the full evolution of the heuristics in a knowledge base is impossible because of the continuous changes and the huge amount of different heuristics and heuristics sets in it, so here we aim at just giving a more practical overview of what happens because of the evolution rules explained in Section 3.1.

First of all, we will show the evolution of the heuristic for deciding whether to activate the `-fgcse-lm` flag of GCC. The functionality of this flag is as follows:

> When -fgcse-lm is enabled, global common subexpression elimination will attempt to move loads which are only killed by stores into themselves. This allows a loop containing a load/store sequence to be changed to a load outside the loop, and a copy/store within the loop.

The following evolutionary sequence is the result of the compilation of a series of programs that are part of a training set as those described in the previous sections of this chapter.

It should be noted that not every compiled program determines an immediate evolution of each heuristic: the evolution happens according to the probabilities listed in Table 3.1 and to the weights of Figure 3.4. Therefore, even if the heuristic hereafter presented was evolved along the compilation of about 90 programs, it is only 5 steps long, because during all the other compilations, either other heuristics in the heuristics set were changed, or the attempted changes did not impact positively on the result and where therefore discarded.

As Figure 3.20 show, at first the heuristic is generated as a constant `false` value.

Afterwards, in Figure 3.21, a mutation influencing the whole formula kicks in, substituting the constant value with a more complex heuristic, including a conditional statement and depending on the average number of basic blocks with less than 15 instructions (`avg_basic_block_less_-15_inst`) and on the average number of edges in the control flow graph (`avg_edges_in_cfg`).

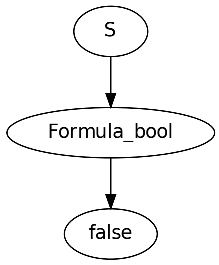


Figure 3.20: The first step of evolution of the `gcse-lm` heuristic.

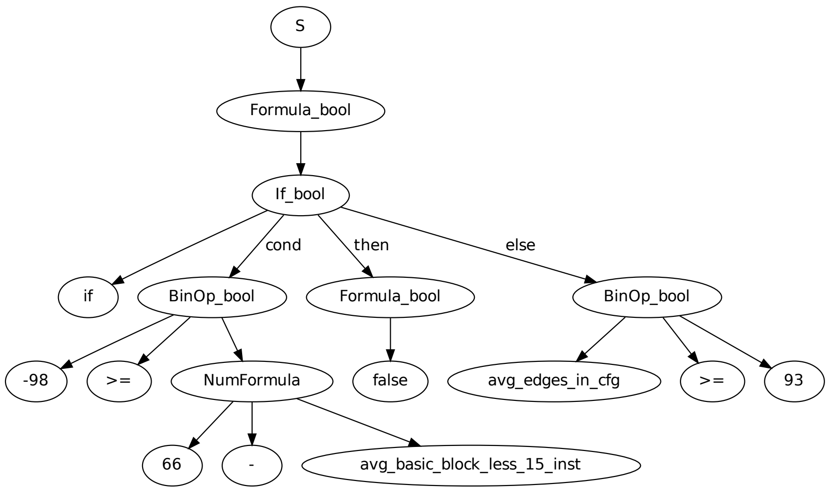


Figure 3.21: The second step of evolution of the `gcse-lm` heuristic.

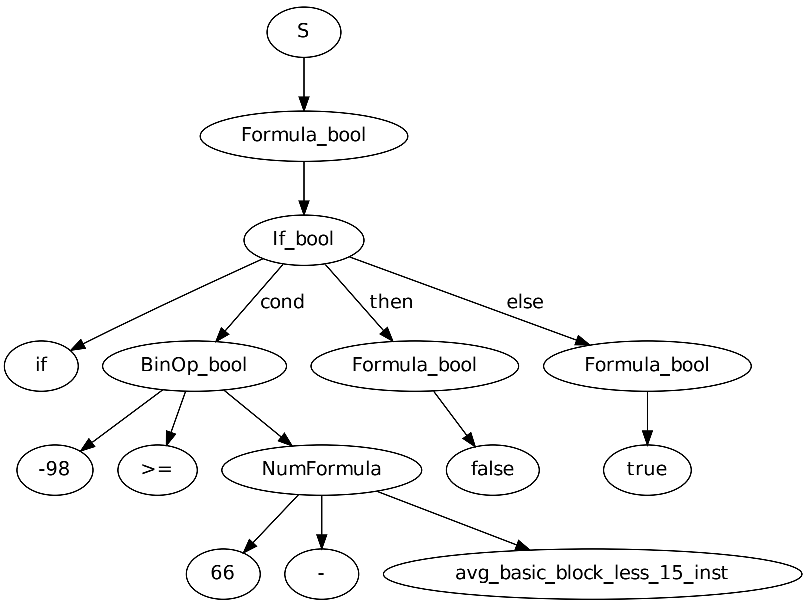


Figure 3.22: The third step of evolution of the `gcse-lm` heuristic.

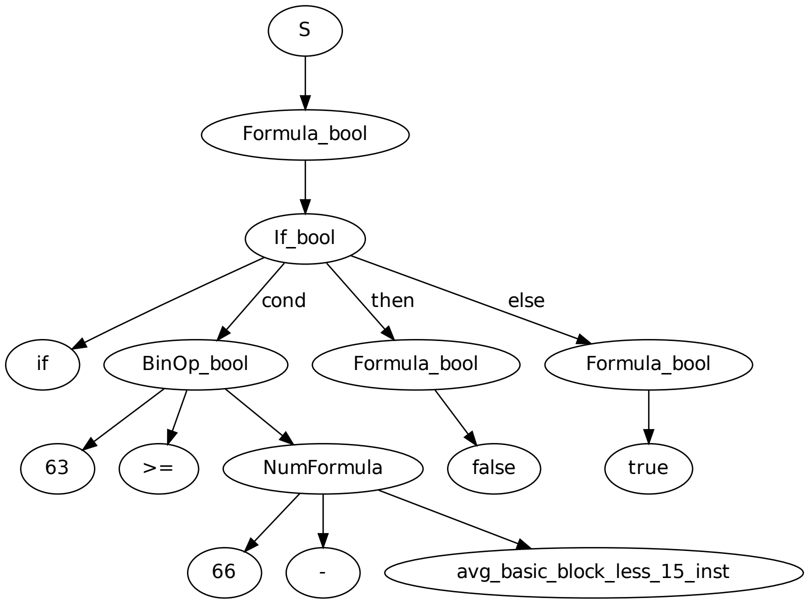


Figure 3.23: The fourth first step of evolution of the `gcse-lm` heuristic.

The next step (Figure 3.22 is again a mutation, influencing the `else` branch of the conditional statement. The binary operation that was there is removed and substituted by a constant `true` value.

Then, an evolution modifies the formula to the form visible in Figure 3.23, with the first numeric value changed from -98 (that used to make the formula be always evaluated to get the `else` branch) to 63. With this change, the formula can now take both branches.

The last evolutionary step (visible in Figure 3.24) is actually composed by two different evolutionary steps. One changes the `then` branch from `false` to `true`, and the other one changes the `else` branch from `true` to `false`. This is an intended side effect of how the long-term learning algorithm is written: it sets a minimum number of differences between heuristics sets, but not a maximum number, so it might happen that a heuristic is actually modified twice in a generation.

It is worth noting that this is just one possible evolutionary sequence for this flag: given that the generation and evolution of the heuristics happens with random techniques, every compiler history will likely be completely different, because different heuristics will be generated, and they will interact in different ways with one another.

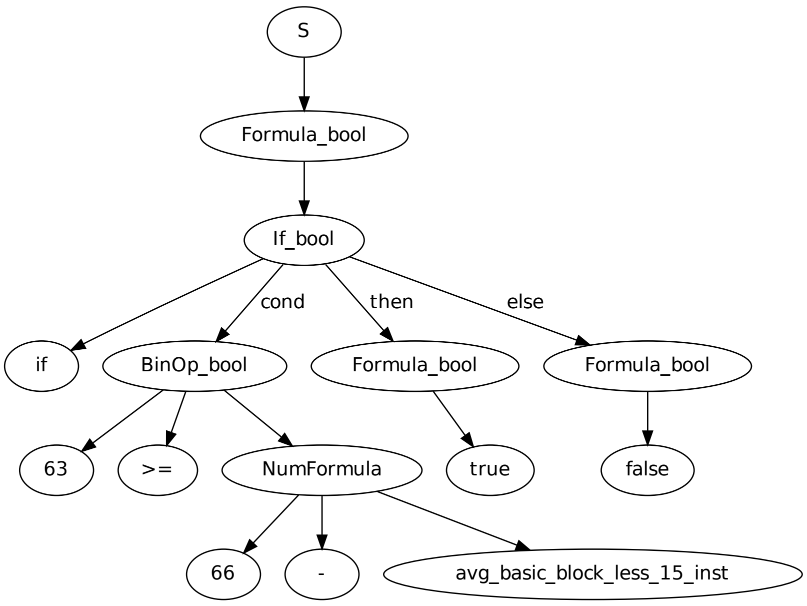


Figure 3.24: The last step of evolution of the `gcse-lm` heuristic.

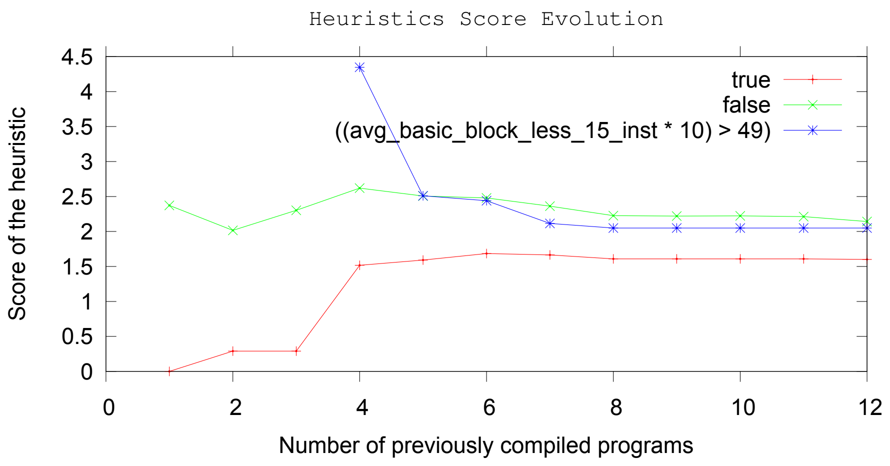


Figure 3.25: The evolution of the score of candidate heuristics over time.

Visualizing the evolution of a full heuristics set on paper is unfortunately not possible, because of the extremely high dimensionality of the problem.

Still, we can at least show the evolution of the score of alternative candidate heuristics for the same role. The graph in Figure 3.25 has been intentionally simplified to reduce the clutter in the image and make it more readable for the sake of this example. It represents the evolution of the score of three different heuristics for deciding whether to enable the `-fweb` flag of GCC, over the compilation of 12 programs. In a real-world version, the graph would contain hundreds of lines (one for each candiate version of a given heuristic) and would extend for a much higher number of compilations (all those made by the compiler during its lifetime). Furthermore, a graph like this could be plotted for every single heuristic that the compiler has to learn. Though, it should be noted that, in order for the knowledge base file to be as small as possible, it does not contain all the information required to plot this graph, because it only stores aggregate versions of these data. In order to plot this graph, an ad-hoc version of the learning-algorithm has been used, able to dump all the data.

In the graph being considered, two of the plotted heuristics are the trivial ones (always true and always false). The third one is a more complex heuristic.

It is interesting to observe that the scores are continuosly evolving, because, in this specific execution, all the three heuristics are actually tested against the programs being compiled, and so there is more information that the compiler can use to learn. In this example, we can observe that when a new heuristic is introduced, it initially behaves particularly well and gets a really high score (so that it will be more likely to generate offspring in the following generations) but then its score is averaged with that of the following compilations, and stabilizes around a lower value. This continuous (but decreasing with time) modification of the scores is what allows the long-term learning to quickly adopt new heuristics that proved good, while at the same time preserving the overall stability.

3.4 Related work

Many researchers have worked towards building compilers without the need for compiler programmers to manually define whether to apply each optimization algorithm.

The authors of [42] exploit periods of stable performance (called phases) that can be found in programs to remove the need for a full execution of a program in order to evaluate a single optimization. Multiple versions of the code for a phase are compiled in the program. Each of the versions is executed for part of the running time, determining which one is the fastest through low-overhead profiling. This allows them to create self-tuning applications. Evaluating optimization performance online is also the aim of [79]. Their system is implemented in a Java VM, and could be used to identify and optimize performance-critical code sections, though using the Java JIT compiler instead of just choosing between multiple pre-compiled versions of code.

[2] show that it is possible to learn a model that predicts good optimizations based on the analysis of static program features. Experimental results from [21] suggest that using performance counters instead of static features better characterizes the program, allowing to further reduce the number of configurations to test.

Another approach was proposed by [33]. Every time they need to compile a program, they build a model by compiling and testing a small number (as low as 8) of versions of the program. Then, they use this model to predict –without execution– the performance of 500 randomly generated points of the search space. Up to 100 of the best points from this set are executed to find the actual best version. The problem with this approach is the need to build a program-specific model, and the fact that it does not reuse knowledge acquired from previous compilation.

Thomson et al. [121] try to reduce the training time of the model by focusing it on the programs that best characterize the search space of the optimization sets. They first gather the static features of all the programs in the training set, then apply unsupervised clustering in the program feature space. This way, a classification of the programs according to the similarity of their features is obtained. The most typical program of each cluster is chosen, obtaining a subset of programs that is representative of the whole search space but much smaller. Each of the programs of the reduced training set is then compiled and tested with 4000 different optimizations sets, using a supervised learning technique to learn the model. The model is then used to immediately provide the supposedly best configuration, one-shot, without using iterative compilation.

The workflow proposed by [84] uses a reverse k-Nearest Neighbors classifier [73] to determine if the program being compiled is similar to a previously met one. If it is, the best configuration that was recorded for the similar program is used, without the need for any test execution. If the program is an outlier, iterative compilation with random search is used to search for good configurations. Information about all the tested candidate configurations is stored. This approach is similar to long-term learning because it does not need a training phase and reduces the number of times iterative compilation has to be used. On the other hand, when it is used, hundreds of executions are needed. Furthermore, it uses a classifier instead of predicting actual heuristics. If the learning process is stopped to increase the compilation speed, heuristics allow outliers to be compiled with good but non-optimal results. This is not possible with classifiers.

The main differences between long-term learning and other approaches are as follows. Most works about machine learning applied to compilers just aim at improving the performance of the compiled programs. Long-term learning, on the other hand, also aims at finding the best compiler heuristics expressed through human-readable formulas, that are easier to understand for compiler writers with respect to classifiers and other more complex models. To the best of our knowledge, the only other works with the same aim are those by [113], using mathematical formulas, and [90], using decision trees. Furthermore, most works learn the single heuristics independently, whereas we also score sets as a whole, capturing the interactions between heuristics. Finally, most works just deal with changing command line flags, whereas our approach is more integrated in the compiler and can change internal parameters of the algorithms as well, as shown with PetaBricks tiling algorithm.

The LLVM compiler implements an only slightly related concept with its lifelong program analysis and transformation [78]. Here, "lifelong" refers to the lifetime of the program ("interprocedural optimizations performed at link-time [...], machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs") instead of the lifetime of the compiler as in long-term learning.

Autotuning [122] is another machine-learning-based approach to improve the performance of programs, and it is also used by PetaBricks [5], on top of which we based one of our implementations.

3.5 Future work

Various directions for improvements are possible, and we mention just a few.

Sometimes, different compiler configurations applied to the same source file might lead to identical binaries [77]. Moreover, since long-term learning is a multi-generation learning algorithm and uses elitism, it is likely that some candidates will be present, identical, in multiple generations. Therefore, we might end up testing the same binary candidate more than once. To prevent this we might compute a hash of the binary file, reusing previously recorded execution times where available. The default parameters of our long-term learning implementation test a total of 18 candidates (as explained in Section 3.3.1) during a compilation. Given that the timing runs are the slowest step of our algorithm, being able to avoid even just one of them, would sensibly speed up the compilation process. This is particularly important when dealing with the candidates in a single generation: they all derive from the exact same knowledge base, so there is a higher probability of generating similar candidates. Furthermore, each generation only has a very low number of candidates, so if two of them are equal, the explorarion of the search space is being greately reduced. The default settings of our long-term learning implementation test a total of 18 candidates (as explained in Section 3.3.1) during a compilation. The timing runs are the slowest step of our algorithm, so, avoiding even just one of them, would sensibly speed up the compilation process.

Currently, the selection of what program features to use in heuristic formulas is done randomly. A statistical approach computing the correlation between features used and the obtained speedup is likely to perform better. Also, adding the ability to compute more static program features, and the ability to use dynamic features derived from performance counters, we could improve the ability of the system to characterize the program being compiled, thus further increasing the quality of the result and the obtainable speedups.

The current version of long-term learning always tests a fixed number of candidate configurations. Although low, this number still might cause quite long compilation times. We could use program features to determine how many candidates to actually test. If we can determine that many similar programs have been compiled before, we could trust the current heuristics and just generate the best candidate accordingly (as in [84]), if the program is completely different from anything known, the full set of 18 candidates should be tested. Intermediate configurations (for example with just 1 or 2 generations instead of 3) could also be used when only some data is available from previously compiled similar programs.

For the work presented in this chapter, we decided to use GCC because, being at the heart of PetaBricks, we could share part of the implementation effort for the two compilers. LLVM [78] could be another interesting compiler on which to implement long-term learning, as it provides a wide variety of optimization passes and the possibility to write a pass manager to decide which ones to execute using our algorithm.

3.6 Conclusion

This chapter presented long-term learning, a novel algorithm to determine the best set of heuristics that a compiler can use to make decisions about which optimizations to enable and what values to assign to parameters, without any human intervention.

Experimental results on multiple architectures confirm that, when implemented in the GCC compiler, the algorithm is able to obtain good results starting from no knowledge at all about good optimizations, reaching and sometimes surpassing the performance of GCC's maximum optimization level. The same holds when the algorithm is implemented in the PetaBricks compiler: the performance we obtain is better than that obtainable by the unmodified compiler.

We also verified that when some initial knowledge is provided by the compiler writer, the algorithm is able to improve the performance, sometimes marginally, sometimes in a significant way.

Also, long-term learning improves over exisiting algorithms by being able to learn sets of heuristics as a whole, instead of single heuristics, and by learning complex decision trees as human-readable formulas. The first feature is important because it allows to take into consideration the interactions of various code transformations, that can be non-trivial: heuristics that are efficient when considered on their own, can yield poor results when in a group because of such interactions. The second feature is important because it enables compiler writers to understand more easily when and how a code transformation should be applied, but without giving them the burden of having to find this out by manual experimentation.

Long-term learning improves the compiler by using the information it gathers from the execution time of the compiled programs. In order to compute it, it needs input data for the programs to be provided. Therefore, the ideal use of a long-term learning compiler is probably inside a test-driven development workflow, where the test data are prepared before the program itself and are already available during the compilation.

Long-term learning is meant to learn compiler heuristics, therefore, it is not strictly required for it to always be active. It allows the user to obtain compiler heuristics targeted at producing good programs for the architecture they will actually run on, not just good programs obtained by the iterations performed while the algorithm is active. Therefore, once satisfying heuristics have been learned, the long-term learning algorithm can be completely disabled, leaving a fast compiler producing optimized programs. Also, given two identical systems, the heuristics learned on one system can be ported on the other immediately giving good results on new programs.

Algorithm 3.1.1: The long-term learning algorithm for compiling one program. Variables marked as *Input* have to be provided by the user of the compiler. Variables marked as *Configuration* are defined by the implementation and are not directly accessible to the user

```
Input: $numGenerations$, number of candidate generations;
       src, source code of the program to be compiled;
       $k$, number of compiler configurations ($S=\{C_1, C_2, \ldots, C_k\}$) to be
       generated. Each is a set of compiler heuristics;
       $d$, input data for the program timing runs;
Output: bin, binary code generated by the best performing heuristics set;
Configuration: $\overline{C}$, set of default compiler heuristics;
               $b$, number of top-scoring candidates to be preserved by elitism;
               $h$, number of candidates to be built with top-scoring heuristics;
               $f$, number of candidates to be built with the most frequently used
               heuristics;
               $N$, set of names of the heuristics that need to be part of a
               configuration;
               $p$, probability of exploring new candidates instead of exploiting
               current knowledge;
               $e$, maximum evolution rate, i.e. maximum percentage of
               heuristics that can be evolved;
               $F$, set of names of the features that can be used in formulas;
               $m$, probability of a mutation to occur instead of an evolution;
               KB, knowledge base including heuristics, sets of heuristics, and
               their usage information;

for $i = 1$ to $numGenerations$ do
    $R \leftarrow \emptyset$;
    $\texttt{bin}_0 \leftarrow$ Compile(src, $\overline{C}$);
    $execTime_0 \leftarrow$ GetExecTime ($\texttt{bin}_0$);
    $speedUp_0 \leftarrow 1$;
    $R \leftarrow R \cup \{\langle \texttt{bin}_0, \overline{C}, speedUp_0 \rangle\}$;
    // Elitism
    $Elite_1 \leftarrow$ GetBestCandidates ($b$, KB, $N$);
    $Elite_2 \leftarrow$ GenerateCandidatesFromBestHeuristics ($h$, KB, $N$);
    $Elite_3 \leftarrow$ GenerateCandidatesFromMostFrequentHeuristics ($f$, KB, $N$);
    $Elite = Elite_1 \cup Elite_2 \cup Elite_3$;
    // Evolution and mutation
    $missingCandidates = k - |Elite|$;
    $NewCandidates \leftarrow$ GetBestCandidates ($missingCandidates$, KB, $N$);
    foreach $C_i \in NewCandidates$ do
        if UniformRandom$(0, 1) < p$ then
            $C_i \leftarrow$Evolve ($C_i$, $e$, $F$, $m$);
    $S \leftarrow Elite \cup NewCandidates$;
    foreach $C_i \in S$ do
        // Generation of new formulas
        $C_i \leftarrow$ GenerateMissingHeuristics ($C_i$, $N$, $F$);
        // Computing the fitness function
        $\texttt{bin}_i \leftarrow$ Compile(src, $C_i$);
        $execTime_i \leftarrow$ GetExecTime ($\texttt{bin}_i$, $d$);
        $speedUp_i = execTime_0 / execTime_i$;
        $R \leftarrow R \cup \{\langle \texttt{bin}_i, C_i, speedUp_i \rangle\}$;
    bin $\leftarrow$ SelectBestProgram ($R$);
    // Updating the knowledge base
    KB $\leftarrow$ UpdateKnowledgeBase(KB, $R$);
return bin, KB
```

Table 3.1: The values we used for the parameters of Algorithm 3.1.1, obtained by experimentation.

Parameter	Value
top-scoring candidates preserved by elitism (b)	1
candidates built with top-scoring heuristics (h)	1
candidates built with most frequently used heuristics (f)	1
needed heuristics (N)	compiler-dependent: see Section 3.2.1 and 3.2.2
exploration probability (p)	0.2
maximum evolution rate (e)	0.3
available program features (F)	see Figure 3.5
mutation probability (m)	0.3

4 Parallel Iterative Compilation: Using MapReduce to Speedup Machine Learning in Compilers

[1] Research has proved that machine learning and iterative compilation techniques can be profitable when applied to compilers to improve the optimizations they perform on programs. Unfortunately, these techniques are hampered by the long training times they require. Even if much research (such as [42]) has been spent in trying to mitigate this issue, most techniques still require multiple compilations and executions of programs at compile time or during a training phase. Testing these candidates in parallel is a good way of shortening iterative and machine learning compilation times.

This chapter shows that parallel execution of multiple training runs can be naturally mapped on the MapReduce programming model and is effective in reducing execution times for iterative compilation. The presented technique allows parallel execution on multiple identical worker nodes or on a single machine by splitting its resources. Experimental results show that an almost-linear speedup can be obtained.

In particular, the main contributions of this chapter are as follows.

We present a novel approach based on MapReduce that is able to improve the performance of iterative compilation techniques by parallelizing them on multiple identical machines. In particular, each machine

[1]Part of this chapter was previously published in [119].

compiles and tests one or more different versions of the program, in parallel with all the other computational nodes. The results of these test runs are later gathered and used to determine which version obtained the best performance.

We also present a method that allows one to exploit the parallelism available in modern machines by applying a variant of the proposed MapReduce approach on a single machine. Using this method, the machine can be split in such a way to allow us to parallelize the iterative compilation of sequential or moderately parallel programs.

Finally, we show experimental evidence of the efficacy of both the approach for multiple machines and the one for a single machine. The experimental results have been obtained through a large and diversified experimental campaign performed on various hardware configurations.

The rest of the chapter is organized as follows: Section 4.1 presents our proposed approach for accelerating iterative compilation with MapReduce. Section 4.2 describes the specific iterative compilation problem used to validate our technique. Section 4.3 shows the experimental results we obtained on diverse hardware configurations. Section 4.4 presents the related work and Section 4.6 concludes.

4.1 MapReduce for Iterative Compilation

In this section we present a new way to speedup iterative compilation, based on the MapReduce paradigm.

What most iterative compilation approaches do is summarized by the algorithm presented in Algorithm 4.1.1. We can consider the sequence of operations comprising the compilation, the execution of enough timing runs, and the scoring of the result of a given configuration as a single task. We name this task the *testing* of the configuration.

The main reason for iterative compilation to be slow is that it needs to obtain multiple versions of the program to be compiled. Each version consists of an executable binary (bin_i in Algorithm 4.1.1) and derives from the same source code (`src`) by applying a different candidate configuration (C_i). Each of the binaries has to be executed multiple times in order to determine the fittest choice. The meaning of "configuration" of a candidate depends on the specific iterative compilation problem at hand: it might be a set of compilation flags, a set of compiler heuristics, a set of numerical parameters, *etc.*

Compiling and running the candidate binaries are time consuming operations. Furthermore, the actual number of candidates to be tested is high, usually ranging in the hundreds. Statistical soundness also re-

Algorithm 4.1.1: Iterative compilation

```
   Input: src, source code;
          $r$, number of runs;
          $k$, number of compiler configurations
          ($S$={$C_1, C_2, \ldots, C_k$}) to be generated
   Output: bin$^*$, binary code generated by the best performing
           compiler configuration

   // Compiles a program through iterative compilation
1  $S \leftarrow$ GenerateCandidateConfigurations($k$)
2  $R \leftarrow \emptyset$

   // Each iteration of the following cycle can be
      executed in parallel by a map worker
3  foreach $C_i \in S$ do
4      bin$_i$ $\leftarrow$ Compile(src, $C_i$)
5      $t \leftarrow 0$
6      for $i = 1$ to $r$ do
7          $t \leftarrow t$+GetExecTime(bin$_i$)
8      $avgTime_i \leftarrow t/r$
9      $score_i \leftarrow$ ComputeScore($avgTime_i$)
       $R \leftarrow R \cup \{ \langle$bin$_i, score_i\rangle \}$

   // The following operation can be done by a reduce
      worker
10 bin$^*$ $\leftarrow$ SelectBestProgram($R$)
11 return $bin^*$
```

quires that each of them is run enough times to smooth out noise from the measuring. Even considering the ideal case, where each of these operations is fast on its own, the process as a whole is slow, and can easily sum up to hours.

Section 4.1.1 will present an approach based on MapReduce to speed up the testing of the candidates on a cluster of identical computation nodes, while Section 4.1.2 will show that a similar technique can be applied to a single computer too.

4.1.1 Using multiple identical nodes

It is immediate to see that the testing of a configuration is completely independent from the testing of every other configuration, therefore we can execute them in parallel.

In particular, it is natural to implement such parallel execution according to the MapReduce programming model, defining the functions as follows:

Map Each invocation of the *map* function receives one candidate configuration as its input. The worker node executing the *map* function takes care of the full testing of the candidate. It compiles the source code applying optimizations and transformations as instructed by the configuration. Then, it runs the generated binary multiple times, summing up the running times and computing their average to smooth out noise. Finally, it assigns a score to the binary, according to a fitness function. This function is specific of the iterative compilation problem at hand, and it depends on the running time of the program and, sometimes, on the generated code size or on the compilation time. The *map* function returns the compiled binary together with its score.

Reduce It is executed by a single worker node. It receives the list of candidate binaries and their scores, and returns the best binary.

The MapReduce paradigm as presented in Google's paper [29] allows the user to specify some functions that perform support tasks (splitter, partitioning, combiner). In the model we propose, there is no need for such functions.

The *splitter* function divides the input data in chunks to be processed by the various *map* nodes. It is useless to us, because the data (that is, the set of candidate configurations) are trivially split: each candidate is a *map* job on its own.

The *partitioning* function takes the output of the map workers and divides it among the various reduce nodes, according to their intermediate key. We do not need it because there is a single reduce node, so, no need to partition intermediate data.

The *combiner* function is executed on the same worker nodes as the map functions. It allows to pre-process some of the intermediate results before they are sent to the reduce nodes, in order to reduce traffic on the network. Usually it implements the same code of the *reduce* function. In our application scenario this function is not needed: even if iterative compilation requires the testing of many candidates, still there are hudreds of them for each compilation process, therefore the amount of data that will actually be sent on the network is negligible with respect to the execution times of the candidates.

Iterative compilation and machine learning techniques are applied to compilation to build a model of complex systems that cannot be easily described by the compiler writer themselves. Therefore, it is obvious that the MapReduce approach we propose can only be applied to an environment composed of identical worker nodes, or, at least, identical map nodes. This way, the compiler is actually able to learn the characteristics of the underlying hardware platform and select the best candidate program, even if the various computations are performed on physically distinct nodes.

The need for identical computation nodes might look like a strong limitation, reducing the applicability of the technique, but it is actually not so. As stated in [113], computing costs are now low enough to make it feasible to dedicate a cluster of machines to searching a solution space. This is even more evident considering that machine learning in compilers has been mostly applied to embedded systems [34]. Embedded systems are strongly resource-constrained, therefore developers try to exploit them as much as possible by applying every possible optimization. Also, programs are likely to be developed for devices that are, or will be, produced in large numbers, so it should be neither difficult nor expensive to use several of them in parallel to speedup the compilation phase. Furthermore, [47] showed that it is possible to learn across different machines, so, as stated in Section 4.6, we could use a similar technique to extend this work with the ability to support heterogeneous clusters.

Section 4.3 will show that the approach can also be beneficial when applied to a computing cluster composed by identical nodes. Here, in the typical setup, an application runs on a single one node, elaborating a part of the dataset. Other copies of the application run on the other nodes, working on a different part of the dataset. Therefore, we can use our MapReduce-based approach to compile the application optimizing it to run on a single node, using multiple nodes to test the various candidate versions in parallel.

In order to perform a fair comparison, all the candidates have to be tested on the same dataset. The test data are generated by (or stored on) the master node, and they have to be delivered to all the map nodes. The easiest way to accomplish this is to use a network filesystem, common to all nodes, such as NFS [102] or AFS [59]. Such an approach guarantees a transparent access to the common dataset, and also to the source files of the program to be compiled.
Nevertheless, the shared filesystem is not strictly required: its functionality could be substituted by the transmission of more data (namely, the dataset and the source files) to all the map nodes as part of the job to be run. They could also be pre-distributed to all the computation nodes

before starting the actual testing, except in the case where the dataset is generated by the master node.

Parallelism is sometimes difficult to exploit for computation because of the overhead introduced by the communication between threads or, even worse, nodes of a distributed computation. In our approach, the overhead due to the transmission of the data of a job over the network connecting the nodes is masked by being executed in parallel by the nodes and it is negligible with respect to the time each node will have to spend testing the candidate. Furthermore, the running times are recorded after the transmission is completed, so it does not influence the testing of the candidates.

4.1.2 Using a single computer

Under certain conditions, the presented technique can also be applied when a cluster of machines is not available.

Nowadays, processors with an increasing number of cores are commonplace, but many programs are written in a sequential fashion, and they do not exploit the parallelism of the machine. Other programs only have a fixed, small degree of parallelism, using no more than a few threads.

The authors of [115] showed that MapReduce is a profitable programming model not just for distributed computation, but also for shared-memory systems. Therefore, we can exploit the existing but unused parallelism of the machine to parallelize iterative compilation.

Given a computer with n times as many computing cores as the maximum number of threads of the program to be compiled, and n times the maximum estimated amount of RAM the program will need during its execution, it is possible to split the computing resources n-fold, and use them to run n *map* nodes, to test multiple programs at the same time.

In order to minimize the interaction between candidates, the best approach is to assign the cores to the programs according to physical connections they have on the CPUs. For example, if a group of cores share the same L2 cache, they should preferably be all assigned to a single running candidate.

Starting multiple MapReduce workers is not enough to ensure that resources are split in the correct way: the operating system scheduler can allocate them according to its own decisions, unless it is instructed to behave differently. As an example, hereafter we present the tools that can be used in a Linux environment to explicitly split the resources among different programs.

According to the specific hardware configuration, two main courses of action can be identified to split the resources among the programs, depending on whether the system has Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA) [56]. NUMA systems are better than UMA systems when it comes to being used for parallel iterative compilation with MapReduce. Each NUMA node is directly connected to its own memory banks, with a private bus between the CPU and the memory. So, as long as each process is limited to use only the RAM that is local to the node it is running on, for our purpose a NUMA system can be considered similar to a cluster of physically different computers with a shared file system.

UMA On UMA systems, we can use the `ulimit` tool to define the maximum amount of memory that is available for the program. The `taskset` tool can be then used to bind the *map* worker process to the right cpu cores.

NUMA NUMA systems provide the `numactl` tool, that can be used to bind a running process to one or more specific NUMA nodes, and to only allow access to the local memory of those nodes. This will prevent most possible interferences with other processes.

The main difference between the Phoenix MapReduce for shared-memory systems, described in [115], and our MapReduce approach, is that Phoenix is implemented on top of the pthreads library, whereas we use the same network-based MapReduce implementation that we use in the distributed scenario. The speed difference of the two implementations has not been measured, since it is beyond the scope of this chapter. The loopback network interface should be fast enough to introduce a negligible overhead. Nevertheless, if our approach is implemented with the aim of being used exclusively in a non-distributed environment, a Phoenix-like implementation could provide better performance, since it is specifically targeted at this kind of systems.

Some works, like [88], try to improve the resource usage while running programs in parallel on machines with shared resources. In particular, they try to detect contention of resources to limit cross-core application interference for latency-sensitive applications and throughput-oriented applications. This kind of approach, altough interesting is not the right one for our goals, because it still leads to 4% of contention on average, which is far too much for being able to detect the small speedups that can be obtained by certain compiler configurations. The MapReduce methodology that we present, with almost no measurable resource contention, is much more suitable for iterative compilation.

4.2 Experimental setup

The description and the evaluation of the specific learning process we used to gather the experimental data presented in Section 4.3 is beyond the aim of this chapter. Nevertheless, for the sake of clarity, we will briefly present the iterative compilation approach we implemented, and provide more specific details about how MapReduce is used in it.

As described in Chapter 1, there are two main research fields dealing with improving the quality of compiler optimizations through the use of computing power: machine learning and iterative compilation.

The problem with machine learning is that, at deployment time, it requires a long training phase (*i.e.* sometimes lasting days, or weeks) to allow the compiler to accurately learn the model it will use to predict what optimizations to apply and how to apply them.

Iterative compilation, on the other hand, does not require an initial training phase, but it needs many different versions of each program to be compiled, tested and compared, in order to find the best one.

We decided to develop a hybrid model, to harness the best of both worlds while avoiding some of their defects. We call this model *long-term learning.*

In long-term learning, as in many machine learning techniques applied to compilation, we give the compiler a way to predict what the best program to be generated is. In particular, we use an approach similar to Meta-Optimization [113]: the model we learn is made of the formulas representing the heuristics to be applied by the compiler. Though, we learn them while we compile the applications, thus removing the offline training phase (at deployment time) that has to be performed by most compilers using machine learning.
This is made possible by an iterative-compilation-like approach. Every time a program is compiled, multiple versions of it are compiled, using different configurations. Each configuration is composed by a set of heuristics. As in iterative compilation, the best program is kept. At the same time, though, the information about which set of heuristics performed better is recorded. The next time a program is compiled, better heuristic sets will be generated, evolving them from this information. This greatly reduces the configuration search space, therefore fewer versions of the program have to be tested with respect to traditional iterative compilation techniques. Also, since we are learning the

Algorithm 4.2.1: Simplified representation of the long-term learning algorithm as implemented in PetaBricks

```
Input: src, source code;
       $n$, size of data matrix to be generated for testing;
       $\overline{C}$, set of default compiler heuristics;
       $r$, number of runs;
       DB, knowledge base including a set of heuristics with the
       corresponding usage information;
       $k$, number of compiler configurations ($S=\{C_1, C_2, \ldots, C_k\}$) to
       be generated. Each is a set of compiler heuristics
Output: bin$^*$, binary code generated by the best performing
        heuristics set;
        DB, knowledge base updated with information collected
        during the execution of the algorithm
// Performs one instance of long term learning through
   iterative compilation.
testData ← MatrixGeneration($n$)
bin$_0$ ← Compile(src, $\overline{C}$)
configFile$_0$ ← Autotune(bin$_0$);
$t \leftarrow 0$
for $i = 1$ to $r$ do
    $t \leftarrow t+$GetExecTime(bin$_0$, testData, configFile$_0$)
$avgTime_0 \leftarrow t/r$
$S \leftarrow$ GenerateCandidateConfigurations($k$, DB)
$R \leftarrow \emptyset$
foreach $C_i \in S$ do
    bin ← Compile(src, $C$)
    configFile ← Autotune(bin);
    $t \leftarrow 0$
    for $i = 1$ to $r$ do
        $t \leftarrow t+$GetExecTime(bin, testData, configFile)
        $avgTime \leftarrow t/r$
        $speedUp \leftarrow avgTime_0/avgTime$
        $R \leftarrow R \cup \{ \langle \text{bin}, C, speedUp \rangle \}$
bin$^*$ ← SelectBestProgram($R$)
DB ← UpdateKnowledgeBase(DB, $R$)
return bin$^*$, DB
```

heuristics (therefore, the model) that produce the best results, when a satisfying level of performance has been reached, the learning process can be completely disabled for all the subsequent compilations, thus

Algorithm 4.2.2: Long-term learning in PetaBricks, with MapReduce applied.

```
Input: src, source code;
       $n$, size of data matrix to be generated for testing;
       $\overline{C}$, set of default compiler heuristics;
       $r$, number of runs;
       DB, knowledge base including a set of heuristics with the
       corresponding usage information;
       $k$, number of compiler configurations ($S=\{C_1, C_2, \ldots, C_k\}$) to
       be generated. Each is a set of compiler heuristics
Output: bin*, binary code generated by the best performing
        heuristics set;
        DB, knowledge base updated with information collected
        during the execution of the algorithm
// Performs one instance of long term learning through
   iterative compilation.
testData ← MatrixGeneration($n$)
bin$_0$ ← Compile(src, $\overline{C}$)
configFile$_0$ ← Autotune(bin$_0$);
$t \leftarrow 0$
for $i = 1$ to $r$ do
    $t \leftarrow t$+GetExecTime(bin$_0$, testData, configFile$_0$)
$avgTime_0 \leftarrow t/r$
$S \leftarrow$ GenerateCandidateConfigurations($k$, DB)
$R \leftarrow \emptyset$
foreach $C_i \in S$ do
    Map (src, $C_i$, $r$, testData, $avgTime_0$, $R$)
(bin*, DB) ← Reduce($R$, DB)
return bin*, DB
```

completely removing the need for multiple iterations.

In the context of long-term learning, a heuristic is a (possibly conditional) mathematical formula depending on some characteristics of the program being compiled, called features. We now provide an example of heuristic. A well-known compiler optimization is loop unrolling: it consists in replicating multiple times the body of a loop in order to reduce the overhead due to checking the loop condition after every iteration. We need to use a heuristic to decide the most profitable number of unrolling to apply to a loop. A heuristic for such a task looks like

Algorithm 4.2.3: MAP function

```
Input: src, source code;
       $C$, compiler configuration;
       $r$, number of runs;
       testData, data to be used for the test;
       $avgTime_0$, execution time for the default compiler
       configuration;
       $R$, result set to be updated
Output: $R$, result set updated with the result of this test
1 bin ← COMPILE(src, $C$)
2 configFile ← AUTOTUNE(bin);
3 $t \leftarrow 0$
4 for $i = 1$ to $r$ do
5     $t \leftarrow t +$GETEXECTIME(bin, testData, configFile)
6     $avgTime \leftarrow t/r$
7     $speedUp \leftarrow avgTime_0/avgTime$
8     $R \leftarrow R \cup \{ \langle \texttt{bin}, C, speedUp \rangle \}$
```

Algorithm 4.2.4: REDUCE function

```
Input: $R$, result set to be updated;
       DB, knowledge base including a set of heuristics with the
       corresponding usage information
Output: bin*, binary code generated by the best performing
          heuristics set;
        DB, knowledge base updated with information collected
          during the execution of the algorithm
1 bin* ← SELECTBESTPROGRAM($R$)
2 DB ← UPDATEKNOWLEDGEBASE(DB, $R$)
3 return bin*, DB
```

this:

`if` $loopNest >= 2$ `and` $loopBodySize <= 4$ `then` 6 `else` 0

where *loopNest*, and *loopBodySize* are features: they are characteristics of the loop for which the unroll factor is being decided. Their values are computed by the compiler and then fed into the formula itself to obtain the unroll factor to be applied to the loop.

Our MapReduce approach to speedup iterative compilation is being developed on top of the PetaBricks compiler [5]. PetaBricks is an open source compiler and programming language developed at MIT that uses machine learning and evolutionary algorithms to autotune [6] programs, by making both fine-grained and algorithmic choices.

No mapreduce	1 Worker	2 Workers	3 Workers
28.79 min	31.14 min	15.26 min	10.48 min

Table 4.1: Compilation times of the matrix multiply benchmark on Hardware Configuration 1 with 16 compiler configurations

PetaBricks programs work on numerical matrices as their input and output data. The compiler produces executables that expose hooks allowing the autotuner to adapt them to the underlying platform and to a given input data size.

Long-term learning (shown in Algorithm 4.2.1) is implemented by modifying PetaBricks with the ability to learn heuristics to take decisions about the optimizations.

Because of how PetaBricks programs work, each candidate binary, built from the same source code with a different set of heuristics, needs to be autotuned before being tested. The autotuning process can be quite long, thus making our MapReduce approach convenient, even if there are less candidates than in a traditional iterative compilation approach. An example of this can be seen in Table 4.1, that shows the testing times for a matrix multiply benchmark (taken from Petabricks benchmarks) autotuned for square matrixes of size up to 1024. When multiple programs are being compiled, the savings can easily sum up to hours, even with just a few workers.

Algorithm 4.2.2 shows the iterative compilation algorithm underlying our long-term learning approach, in its MapReduce form (Algorithms 4.2.3 and 4.2.4 show the Map and Reduce functions respectively). It is easy to see that it is an instance of the general iterative compilation algorithm presented in Algorithm 4.1.1. The main differences are the need for autotuning after each binary is generated, the presence of a knowledge base storing the long-term learning data, and the need for a "default candidate" to compare against.

Every time an instance of long-term learning runs, our MapReduce approach is used to parallelize the testing of the candidate heuristic sets.

It is worth noting that in this application of the approach we propose, the fitness function is the speedup of the program compiled with each set of heuristics. The speedup is computed with respect to a version of the program compiled with a default, fixed set of heuristics. This version is tested prior to starting the MapReduce job.

4.3 Experimental Results

This section presents the experimental results that show the profitability of the approach we presented in Section 4.1. The exact application we used as the experimental setup to gather the data hereafter presented is the one we detailed in Section 4.2.

In order to show the wide applicability of our approach, we used a variety of hardware platforms and setups. For each of them, we ran and timed a series of executions, varying the number of heuristic sets to be tested, and the number of worker nodes we used. The minimum number of sets of heuristics is 4, because it is the minimum number that can be used by our long term learning approach (because of the elitism mechanisms described in 3.1.4). The maximum number of heuristics varies for each hardware configuration. It depends on how many executions we could run given the constrained computing time available on the machines. The maximum number of worker nodes depends on the underlying hardware platform. Every point of data on the graphs represents the average running time of three different run, in order to reduce the effect of noise on the measurement.

Our long-term learning framework is written in Python, and the MapReduce implementation is based upon the MinceMeat.py library [36]. The library is open source, and we improved it by adding the possibility of manipulating multiple sets of data without the need for the worker nodes to establish a new network connection to the master node. This was only for the sake of making the usage of the library more practical, and does not have any impact on the performance of the original library.

Here are the hardware configurations we used to run the tests:

Hardware Configuration 1 3 different machines, used as 3 worker nodes. Each machine is equipped with four Intel Xeon X7550 processors (8 physical cores each) running at 2.00 GHz and 128 GB of RAM.

Hardware Configuration 2 1 machine, used as 4 worker nodes. The machine is equipped with four AMD Opteron 8378 processors (4 physical cores each) running a 2.40 GHz and 32 GB of RAM. Each processor is a NUMA node with four cores and 8 GB of local RAM, and is used for running a MapReduce worker node.

Hardware Configuration 3 1 machine, used as 2 worker nodes. The machine is equipped with an Intel Pentium D processor (2 physical cores) running at 2.80 GHz, with 2 GB of RAM. This is a UMA system, so each worker node is statically assigned one of the cores, and 1 GB of RAM.

Hardware Configuration 4 10 different machines, part of the CILEA Lagrange cluster [7], used as 10 worker nodes. Each machine is equipped with two Intel Xeon X5460 processors (4 physical core each) running at 3.16 GHz and 16 GB of RAM.

The graph in Figure 4.2(a) presents the results for Hardware Configuration 1. Figure 4.2(b) is Hardware Configuration 2 and Figure 4.3(a) is Hardware Configuration 3. Finally, Figure 4.3(b) shows the performance of Hardware Configuration 4. The behavior on all the Hardware Configurations is comparable: every instance of the long term learning algorithm we run includes a sequential part, namely the generation of test data and the compilation and testing of the program compiled with the default set of heuristics. Then, MapReduce partitions the testing of all the candidate programs among the available map worker threads, in parallel. Therefore, we can compute the maximum theoretical speedup according to Amdahl's Law [56]. Given c compiler configurations to test and N worker nodes, we consider the sequential part[2] to be $S = 1/(c+1)$ and the parallel part to be $P = c/(c+1)$. The results we obtain (shown in Figure 4.1(a)) are close to the theoretical speedup $s(N) = \frac{1}{S+P/N}$. As shown in Table 4.3 and Figure 4.1(b), the influence of the sequential part becomes less important with the growing number of heuristics, allowing our approach to scale almost linearly with the number of available worker nodes.

The results of Figure 4.3(b) are less regular than all the other results. This is due to the network filesystem connecting the nodes, that is shared by all the nodes of the cluster. Processors and RAM of the nodes can be requested for exclusive use to run the tests, but the filesystem is common to all the nodes of the cluster. In our experimental setup the recorded times include the time the filesystem takes to transfer the data to the computation node. Because of this, other users of the cluster could affect the measurement.

It is worth noting that the time we measure to plot these graphs is the total execution time of the iterative compilation, because we want to show the speedup that can be obtained through MapReduce for the whole process. On the other hand, the time measured for determining

[2]The sequential part consists in the compilation and execution of the candidate obtained by using the default configuration that will be used as a reference. This is executed before the parallel execution of all the other c candidates starts.

Configuration	Mean	Variance
Single core	66.5124 s	0.014
Parallel (core 1)	66.7308 s	0.019
Parallel (core 2)	66.6710 s	0.014

Table 4.2: Time needed to complete the testing of the exact same candidate (recorded on 30 repetitions) on Hardware Configuration 3, run with both cores in parallel, or on a single core

how good each candidate is only includes the running time of the candidate: the timing starts after the data have been moved locally, therefore is not influenced by the filesystem performance. This way, the candidate comparison is more fair.

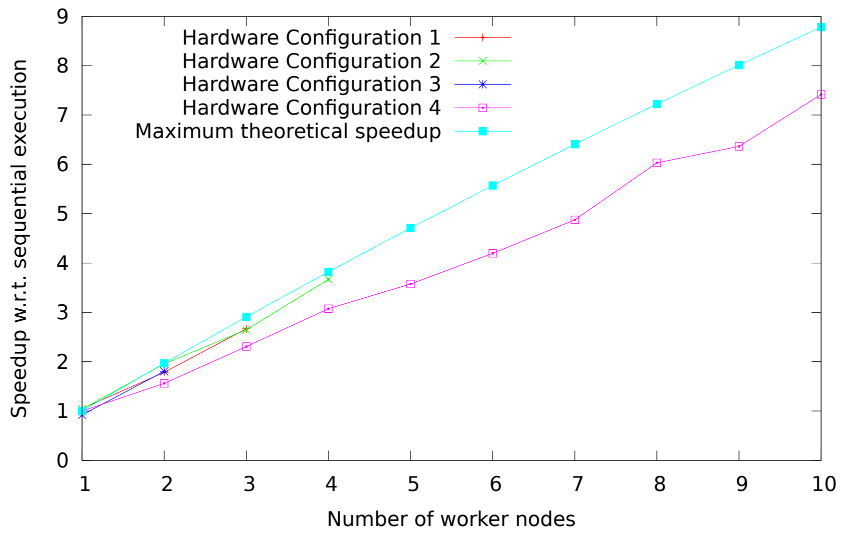
Figure 4.1(a) shows the speedup that can be obtained when compiling a program with 64 different sets of heuristics on the various hardware configurations. Hardware Configuration 4 has the less linear speedup, because its number of nodes is big with respect to the number of compiler configurations to be tested. Figure 4.1(b) shows that, given a fixed number of nodes the more configurations are tested, the closer to linear the speedup is.

To show that there is no interference between tests executed in parallel on a single machine by splitting the available resources, we performed another experiment: we picked a program, and tested it on Hardware Configuration 3, the more constrained we have, that is more likely to show interferences between parallel testing if they exist. For 30 times, in parallel on the two cores, we compile the program with the default set of heuristics and autotune it. Then, we do the same on a single core, leaving the other unloaded. Being the exact same program with the same configuration, we expect the results to be very similar. The actual results can be seen in Table 4.2: the timings for the single core execution of this benchmark are comparable with those of the two parallel execution, confirming there is no interference.

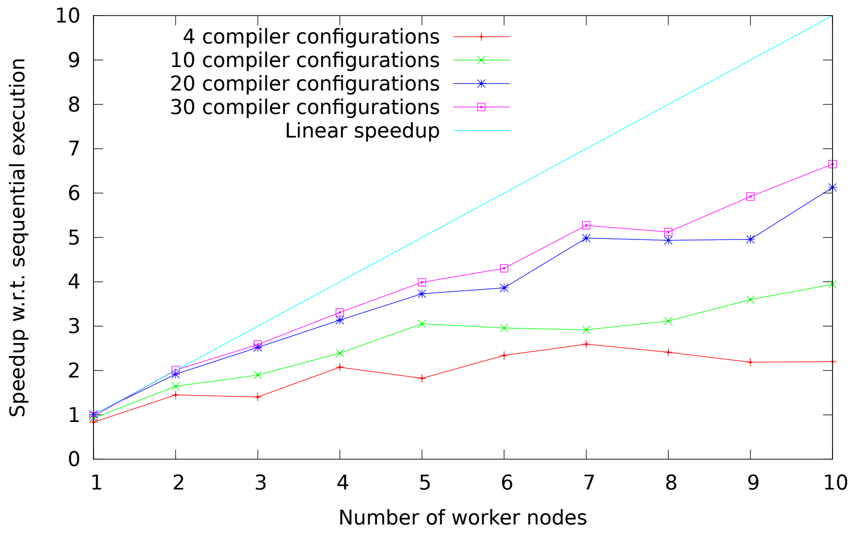
4.4 Related Work

In the field of machine learning some research has been carried on aiming at exploiting parallelism to speed up the learning algorithms.

Low et al.[85] observe that designing and implementing efficient, provably correct parallel machine learning algorithms is challenging. Therefore, they propose GraphLab, a framework that improves common pat-

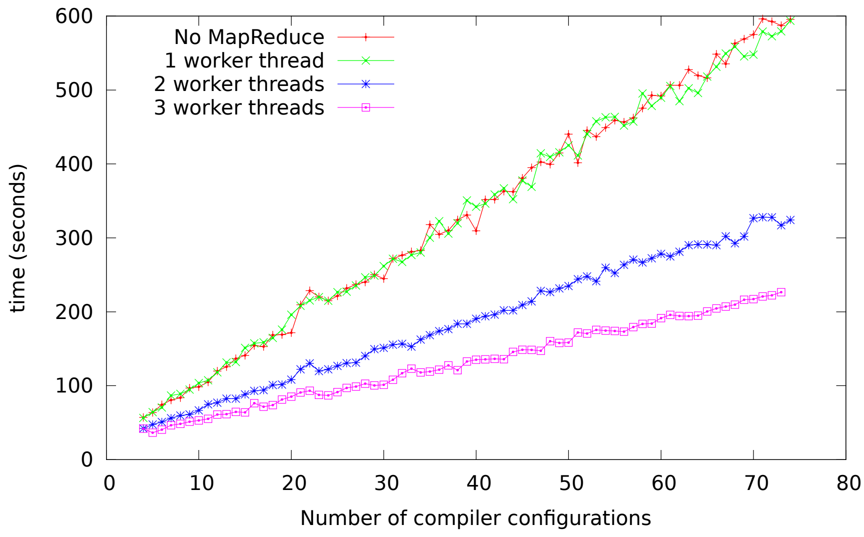


(a) Speedup over using a single worker node for 64 compiler configurations. The maximum theoretical speedup is computed according to Amdahl's law.

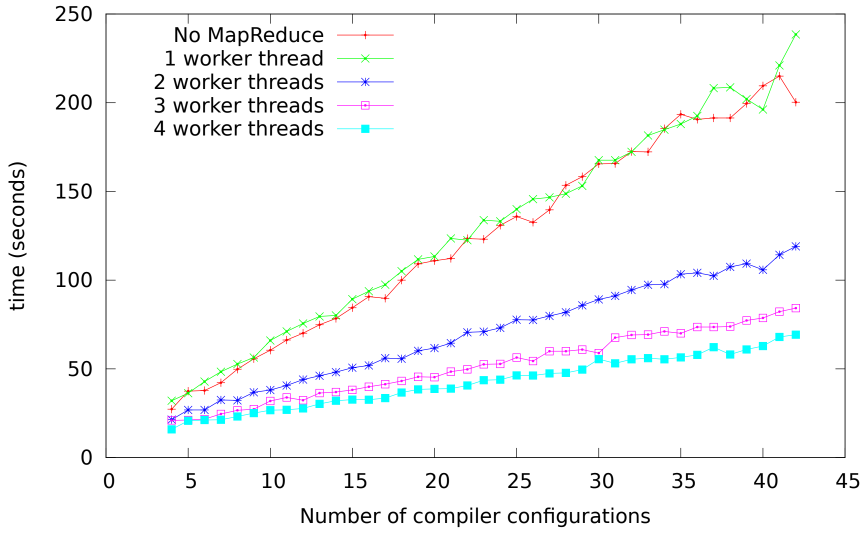


(b) Speedup over using a single worker node for varying numbers of compiler configurations

Figure 4.1: Speedup obtained with the presented approach

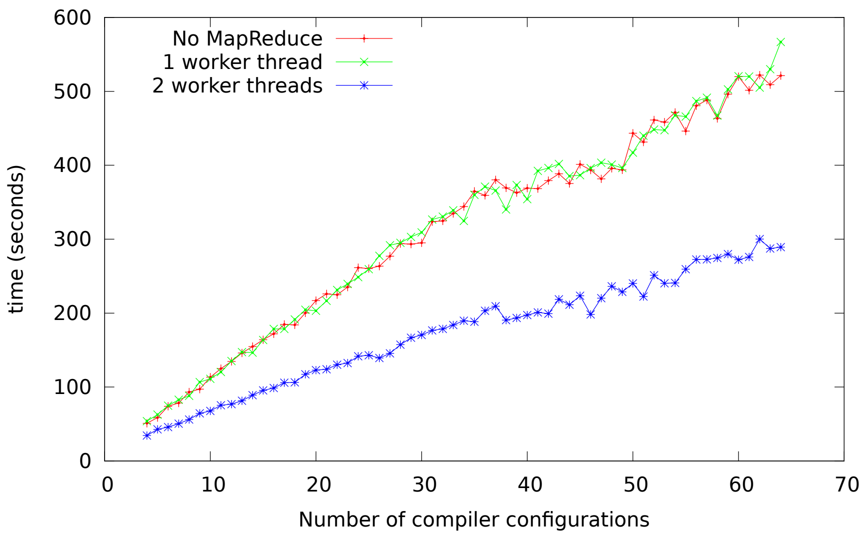


(a) Hardware Configuration 1

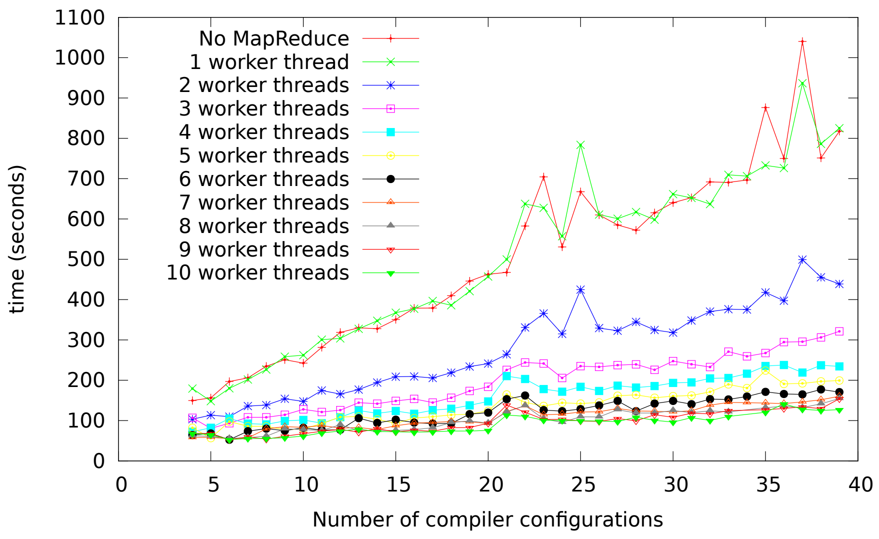


(b) Hardware Configuration 2

Figure 4.2: Execution times on various hardware configurations



(a) Hardware Configuration 3



(b) Hardware Configuration 4

Figure 4.3: Execution times on various hardware configurations

Tested compiler configurations	2 Worker	3 Workers
4	74.01 %	73.43 %
10	67.62 %	53.78 %
20	63.04 %	49.68 %
30	61.75 %	41.36 %
40	61.53 %	43.63 %
50	53.34 %	35.96 %

Table 4.3: Time needed to complete the compilation on Hardware Configuration 1. Percentage with respect to execution without MapReduce

terns in machine learning an parallelism, such as MapReduce, and enables machine learning experts to to easily design and implement efficient scalable parallel algorithms by composing problem specific computation, data-dependencies, and scheduling.

Much research effort has been spent in making compilers able to extract parallelism from sequential programs [17, 100].

Not as much work has gone into parallelizing the compiler itself, probably because this is usually a less important issue: most of the effort in this area was about Just-In-Time compilers, for hiding compilation latencies at runtime [19, 74]. The authors of [127] use different machines in parallel to prepare different variants of code to be executed, using runtime profiling, dynamic code generation and pruning of the search space to identify the most promising code variants to use.

MapReduce, in particular, has not been used much with compilers. To the best of our knowledge, the only work in this direction was MRCC [86]. MRCC is "an open source compilation system that uses MapReduce to distribute C code compilation across the servers of the cloud computing platform", on top of Hadoop [120]. This means that it computes the dependency graph of the various C source files composing the program and then uses MapReduce to compile multiple independent files at the same time. Such an approach is completely different from the one we propose, that is based upon machine learning and performs multiple compilation of the same files with different parameters. Therefore, we do not present a timing comparison of MRCC and our system, that would be meaningless.

In the area of iterative compilation not much research has been dedicated to parallelization as a way to reduce the execution times. The authors of [60] and [21] use parallel execution of tests on a cluster to validate their work, though without using MapReduce. Section 4.5 presents a few reasons why this is an important topic to consider and why MapReduce is a good choice for implementing it.

On the other hand, we can identify some other work looking for ways to reduce the time needed to perform the iterative compilation process.

Most of them focus on trying to prune the search space, thus reducing the number of candidates that need to be tested. Nisbet [96] uses a genetic algorithm to guide the candidate selection process. Kisuki et al. [68] use a simple algorithm based on recursively refining a n-dimensional coarse grid over the search space. Kulkarni et al. [77] introduce a number of techniques, mainly based on memoization, to avoid running again candidates that have already been tested.

Fursin et al. [42] try to reduce the time needed to test the candidates themselves. Instead of performing a full run for each candidate, they suggest identifying phases of stable performance in the program. For each phase, multiple versions of the code are generated in a single executable file. At runtime, the various candidates are tested during a single execution of the program, and the best one is chosen.

4.5 Rationale

Parallelizing iterative compilation algorithms might seem like a straightforward problem. In fact, many papers on the topic of iterative compilation dedicate just a couple of lines to stating that the tests are executed on multiple machines, or do not say anything at all about that. This is likely due to the fact that the basic idea of testing different candidate configurations in parallel is in fact a so-called embarassingly parallel problem, with no need of complex synchronization procedures between the various parallel jobs.

The problem is that there is no standard for this kind of task, therefore every researcher has to come up with its own solution from scratch. Defining a standardized procedure for dealing with this problem would allow reasearchers to implement and share a common framework, focusing on more interesting parts of the compilation process.

The aim of this chapter is to propose such a framework, built on top of an existing, well known and widespread technology, namely MapReduce.

The choice of MapReduce is due to the following reasons. First of all, it fits perfectly the problem at hand. The *map* function can be used to test the various candidates and the *reduce* function can select the best result of the test runs. Moreover, MapReduce is a well known programming paradigm, easy to understand and really widespread. This makes it easier to find ready-made libraries implementing it for many programming languages, reducing the burden on the compiler writer. Such libraries can provide features for dealing more efficiently with the job queues, distributing the compilation and testing tasks to the free *map* nodes, thus minimizing the running time, with no time spent writing an home-brewed version of such algorithms.

Furthermore, many existing computing clusters, both academic ones and commercial ones, already support MapReduce, so it is particularly easy to find the computing resources where to use a system implemented according to such a paradigm.

On top of this, this chapter also aims at showing that such an approach can be a practical way to better exploit the computational resources of a single machine, when it is powerful enough, by splitting them. The chapter presented an example of how to do this on UMA and NUMA machines, discussing the benefits and shortcomings of the two approaches, and showing that the results obtained on a variety of hardware configurations confirm the viability of the approach.

Being able to split the resources of a single machine with no side effects on the obtained measurements could lead to a great reduction in the execution times of many iterative and machine learning compilation approaches.

This should not be seen as an alternative to other approaches to reduce these times, such as the one described in [47]. Actually, our MapReduce algorithm is studied to fit well with any approach requiring multiple compilation and executions, and could be used to further improve most of them.

Particularly interesting is the comparison with Collective Optimization [47], that aims as well at parallelizing the burden of training the model. This comparison cannot be a quantitative one, but only a qualitative one, because the approaches are too different: Collective Optimization divides the task of building the optimization model across different users over time, whereas our MapReduce approach aims at using multiple machines at the same time.

The capability of Collective Optimization to learn across different hardware configurations is its strongest advantage, and it could be extremely useful to implement a similar approach on top of MapReduce, to avoid the limitation of requiring an homogeneous computing cluster. The two metodologies could therefore be used together with good results.

Another method for expediting the evaluation of multiple candidate versions of the code is described in [42]. The paper describes a method for exploiting the structure of particular programs that present relatively stable phases to compare multiple versions of the code in a single run. Of course, each executable needs to already contain the various versions of the code to compare, thus leading to an increase in its size. Therefore, even this approach could be compatible with the proposed MapReduce methodology. By creating multiple executables, each with some code versions, it would be possible to test many of them without the need to increase to much the size of each one. It should also be considered that MapReduce on its own does not require any particular program structure, whereas phase tuning requires the program to exhibit phases. So, if the two approaches are not used together, we can say that phase tuning is good for parallel programs, using many computational resources, as long as they have phases: this way, a single executable can be run a single time on its own on a computer in order to find the best candidate. On the other hand, MapReduce is better for applications with a low degree of parallelism, where the computational resources (and particularly, the processor cores) of a computer can be split to allow multiple programs to run together.

4.6 Conclusion and Future Work

We presented a novel approach, based on the MapReduce programming model, that allows one to speed up the execution of iterative compilation, through parallel execution of the tests that are usually executed sequentially but are independent and can therefore be run at the same time. The experimental campaign on four different hardware configurations showed that the speedup is almost linear with the number of worker nodes dedicated to the execution of the tests.

Taking into account more specific issues of compilation, we can envision new improvements to further increase the performance of the system. It is known that different sets of optimization can happen to generate the same compiled program [77]. Therefore, we could implement a two-phase MapReduce setup. In the first phase, the *map* function compiles the program with the given optimization settings and computes a hash of the generated executable file. Then, the *reduce* function compares the hashes, removing duplicates. In the second phase, a new *map* function executes the timing runs for each non-duplicate program and computes their scores, whereas the new *reduce* function selects the best one.
Another improvement could come from changing the way new heuristics are generated: instead of immediately generating all the possible configurations to be tested, we could generate only a subset of them, of the same cardinality as the number of map workers (or a multiple thereof).

We could then test these candidates, and generate the next ones by evolving the best scoring ones in this set, until the desired number of tested candidates is reached.

Fursin and Temam [47] showed that is is possible to acquire knowledge about program optimization using data coming from a wide variety of different hardware platforms, while at the same time removing the need for a long training phase. They did this by building a set of probability distributions that correlate the behaviour of different architectures. One probability distribution is generic and enables optimizations that have proved to work well in general. One is built by using information from architectures that proved to behave similarly for a given optimization algorithm, according to the test run that have already been executed. The last distribution is specific for the architecture at hand and is learned when enough samples have been provided, by compiling enough programs. Using probability distributions akin to the one used for collective learning in [47] we could allow the users to exploit clusters of non-homogeneous machines to perform the parallel testing of the various versions of the program being compiled. This way, we could further widen the applicability of our approach by removing the need for the worker nodes to be identical machines.

The presented approach is not mutually exclusive with other methods meant to speed up the process of iterative compilation. Machine learning or other methods can be used used to prune the search space of the possible solutions in advance, or to focus the search in a given direction, thus reducing the number of candidates to be tested, by excluding those that are likely to perform poorly. At the end of the search process, when the candidates have been selected and they need to be tested, our approach can be applied to speedup their testing through parallel execution. This cannot be done when the generation of each candidate depends on each of the previous ones, but it is applicable in every case where a set of candidates is generated at the same time and then they can be tested in parallel.

5 Future work

This chapter will present some examples of ongoing works that could benefit from using techniques derived from or built upon long-term learning.

Such works have already been published in international venues, excluding the part regarding long-term learning itself.

5.1 A Lightweight Approach to Compiling and Scheduling Highly Dynamic Parallel Programs

[1] This section describes a dynamic and lightweight compilation approach able to shepherd the execution of highly dynamic parallel programs at runtime without the need for a full-fledged Just-In-Time compiler.

The proposed approach allows one to apply profitable optimizations that could not be used at compile-time due to lack of sufficient information. It also enables one to optimize code fragments multiple times, depending on the specific conditions of the execution environment at runtime, while limiting the performance overhead to a negligible amount.

The section will also show the importance of the techniques developed in Chapter 3 for the lightweight approach we hereby present.

In particular, Section 5.1.1 introduces our approach to perform runtime compilation using lightweight compilation micro-threads and run-

[1]Part of this section was previously presented as a poster in HotPar '12 [111] and was developed in cooperation with my colleague Ettore Speziale.

time scheduling. Section 5.1.4 discusses scenarios where our technique can be useful and Section 5.1.5 concludes.

5.1.1 Proposed Approach

In this section we present a new kind of compiler optimizations, able to adapt to highly dynamic execution environments without adding excessive overhead at runtime.

Optimizations built according to our approach are divided in two phases, one to be executed at compile time and one at runtime. The runtime phase is extremely lightweight and is assigned the task of modifying the program to actually apply the optimization according to the current state of the execution environment, whereas the compile-time phase has to generate the machine code of the program in such a way to allow this to happen

The need for offloading most of the optimization-related computation on the static compiler has already been assessed by other works, such as [97]. Another example of cooperation between compiler and runtime can be found in [53] for GPUs.

With respect to the traditional static/dynamic compilation flow, where compilation and execution phases are clearly separated, we have to face two specific issues.

Expected profitability: not all optimizations have to be delayed at runtime. We aim at applying an optimization at runtime only if *there are no sufficient information* to apply it at compile-time and a *considerable improvement* is expected. At the same time, since code is generated at compile-time, we free the runtime environment from the burden of applying trivial but needed optimizations, such as copy propagation, that a traditional JIT approach has to perform during program execution.

Moreover, delaying at runtime all applicable optimizations is not feasible, because we aim at keeping a lower overhead with respect to traditional JITs. This naturally leads to a careful selection of which optimizations to delay, based on their expected profitability, as described in Section 5.1.2.

Compiler interference: runtime application of optimizations leads to possible conflicts between optimizers and the running optimized code. This happens because there is no strong separation between the compiling and running phases of the program. To guarantee consistency, it is necessary to coordinate optimization and execution of the code.

To handle these issues, we define a model that allows to detect, handle and apply profitable optimizations. We represent the program using a set of micro-threads (similar to those described in [35, 39, 65]). Part of these micro-threads are defined by the programmer or by the compiler and contain the code of the program being written. We call them *computational micro-threads.* The remaining micro-threads are called *compiler micro-threads.* They are generated by the compiler and contain the code that is able to apply optimizations at runtime.

Each compiler micro-thread is associated to a computational one, and manipulates one of its *optimizable region*s, that are the sections of the code of a computational micro-thread that can be modified by a runtime optimization. The dual of an optimizable region is an *optimizing region.* It is defined as all the code of a computational thread that is not part of the corresponding optimizable region. The optimizing region is the region where the optimizer micro-thread can safely run concurrently with the computational micro-thread to apply its optimization.

Compilation/Execution Pipeline

With reference to Figure 5.1, our compilation approach is split into two parts: *generation of compiler micro-threads* and *runtime optimization.*

The first step is intended to be part of a static compilation pipeline, and its goal is preparing the code to be optimized at runtime. We want to consider only optimizations that cannot be applied at compile-time, so this step should be run after standard compiler optimizations. First of all, it has to look at the input code to find candidate applicable runtime optimizations. It is not possible to apply all optimizations, because interferences between them are possible. Therefore, they must be scored with respect to the expected profitability (according to one of the methods presented in Section 5.1.2). Then, the model based on optimizable/optimizing regions allows to represent such interferences on the computational micro-thread control flow graph. A pre-scheduler analyzes the interferences and selects the best optimizations. Finally, the corresponding compiler micro-threads are generated from a library of *micro-optimizers.* For each computational micro-thread, multiple compiler micro-threads can be generated, one for each optimization.

It is worth noting that the micro-threaded model is a purely logical one: we aim at minimizing runtime overhead, therefore if the system is implemented on a computing architecture with high costs of inter-thread communication the micro-threads can be multiplexed into a single mixed micro-thread. To do this, the pre-scheduler analyzes the computational micro-thread and compiler micro-threads, and schedules in a single mixed micro-thread the instructions from both, according to constraints imposed by optimizable and optimizing regions. Merging

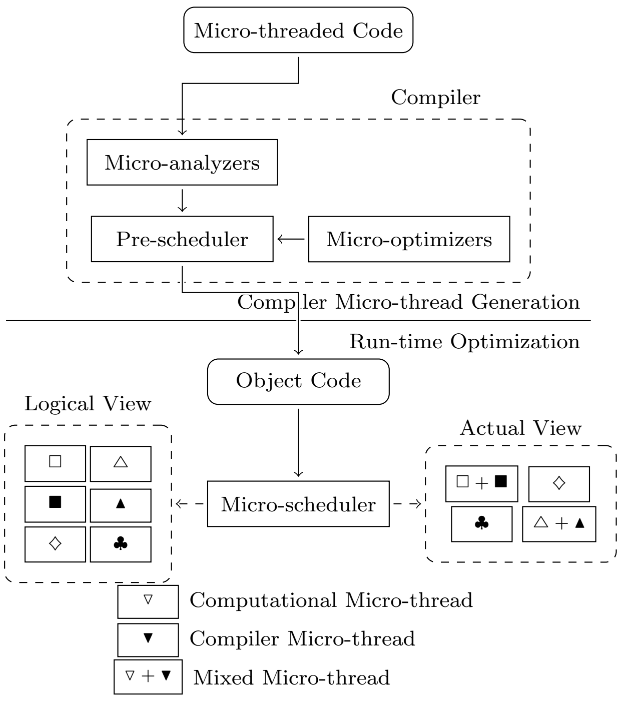


Figure 5.1: proposed compilation/execution pipeline. Micro-threaded code is analyzed to detect profitable runtime optimizations. Compiler micro-threads are built and possibly merged with computational micro-threads

different micro-threads together was proven to be effective for scheduling Single Instructions Multiple Threads programs [83, 82, 114]. In our approach, micro-threads are not homogeneous, but we think that similar techniques have to be used to limit as much as possible the overhead of runtime optimizations.

The output of the pre-scheduler is a set of threads containing micro-threaded code intended to be run by a runtime micro-scheduler. From the logical point of view, the runtime scheduler has to manage both computational and compiler micro-threads, but, due to pre-scheduling, it actually has to manage mixed micro-threads too: therefore, some of the micro-threads need synchronization, whereas some other micro-threads have already been merged by the pre-scheduler, thus eliminating the need for explicit synchronization.

5.1.2 Determining profitable optimizations

Deciding what optimizations can be profitably applied at runtime is of paramount importance for the approach we are proposing. At the same time, unfortunately, this is an extremely difficult task: the complexity of modern computing architectures makes it impossible to fully model them to predict the exact outcome of applying an optimization algorithm to the code of a program. This holds even more when multiple interactive optimizations have to be considered. It is obvious that deciding which optimizations are likely to provide the best outcome when applied at runtime is at least as difficult.

A similar problem has been faced before for selecting the set of optimizations to use during the compilation process of a program. Therefore, we could exploit all the research that was directed at finding a solution for that issue.

In particular, machine learning techniques have been shown to be able to provide a valid solution.

Classic machine learning techniques

Dubac et al. [33] show a technique that is able to predict the speedup that an optimization algorithm can provide, with a good degree of precision. First they train a predictor, by taking a set of training programs and correlating some static features (extracted by analysing their source code) with the effect of various optimizations applied to the programs.

Then, every time a new program needs to be compiled, they compute its static features and they show that they are able to correctly predict the speedup obtainable by applying every optimization the predictor was trained on.

In an analogous way, we could predict the expected speedup obtainable by selecting an optimization for being applied at runtime with our lightweight compilation approach.

Even better decisions about what optimizations to apply at runtime could be taken by using a predictor trained on dynamic features of the program (such as performance counters) instead of just static features, as suggested by Cavazos et al. [21]. This is due to the fact that dynamic features better describe the behaviour of the program at runtime, therefore they would be particularly apt for our objective of deciding which optimizations could benefit the most from being postponed.

Long term learning

The kind of predictor described in the previous paragraph needs a long training phase, to be executed offline. This can be acceptable in general, but sometimes it is important for the system to be able to adapt dynamically to a continuously changing environment runtime environment, or to improve its ability to deal with a specific kind of programs that is compiled more frequently than others.

In such a case, an approach similar to the one presented in Chapter 3 is better.

It is worth noting that such an approach does not just find the single optimizations that perform well when postponed at runtime, but it is implicitly able to determine the best set of heuristics that perform well when used together. This is due to the way we score single heuristics and sets of heuristics, as explained in Section 3.1.6

Deciding whether to postpone the optimizations singularly The decision about whether to apply an optimization at runtime can be trusted to one heuristic. The system will have to learn one heuristic for each optimization algorithm.

The variables used inside the heuristic formulas will be features of the program, either static or dynamic. One reason why we might postpone an optimization to runtime is when a value we need to use to decide how to apply the optimization itself is not available at compile time. Therefore, we will also provide the heuristic generators with features indicating whether certain values, used by the optimizers, are available or not.

In case the optimization cannot be applied without a certain value, this will force it to be applied at runtime when the value is not available. On the other hand, if the value is useful but not required, the generated heuristics will take care of evaluating whether it is better to postpone it or not, likely by using the knowledge about the availability of the value.

Determining the expected profitability Another possible approach based on the techniques described in Chapter 3 is to have the heuristic generator determine a set of heuristics, one for each optimizer: by evaluating each of these heuristics for the specific program we are dealing with, we will obtain a value for each of them, representing the expected profitability. The values will be used to determine a ranking of the heuristics.

The highest scoring ones will be postponed at runtime, the rest will be kept at compile time.

Given a fitness function that awards the better performing programs, the heuristics will automatically evolve in such a way to give a higher score to those programs that benefit the most from being optimized at runtime.

5.1.3 Run-time Optimization

The compiler micro-threads have to change the code of their associated computational micro-thread to optimize it. This can be done explicitly, using *self-modifying code*, or implicitly, using *branch tables*.

The compiler micro-thread is generated together with the optimizable region code it is associated to. Indeed, knowing the layout of the optimizable region, it is possible to generate instructions performing binary rewriting at runtime, without influencing other regions of code of the computational micro-thread.

The strength of self-modifying code is the ability to generate the most suitable instructions for a given optimizable region. However, the cost of code morphing is considerable. An entire region of code must be rewritten. This requires editing the memory locations that store the optimizable region. Moreover, if the code is shared by multiple micro-threads, code cannot always be modified: the conditions triggering runtime optimization for a given micro-thread could be not valid for the others. Despite these limitations, self-modifying code can be an effective optimization strategy, if exploited for highly profitable optimizations, like inner loops vectorization.

A branch table, on the other hand, is a collection of unconditional jumps to different locations. At runtime, an index is used to select where to jump to. It can be implemented using different techniques, and is used to translate switch statements or to implement virtual tables. In our context, branch tables enable compiler micro-threads to change the execution flow of the associated computational micro-thread without changing its code. When our logical model is implemented, an optimizable region should be represented as a collection of sub-regions linked using branch table-based jumps. Compiler micro-threads just have to modify the indices used to select the active jump in branch tables, thus implicitly modifying the control flow graph of the computational micro-thread.

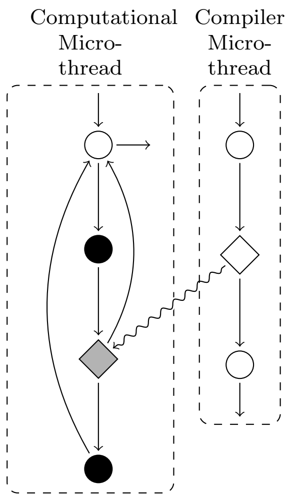
With respect to self-modifying code, branch tables impose less runtime overhead, since applying an optimization simply amounts to setting a set of indices. On the other hand, all the possible fragments of code used to optimize the region need to be generated at compile-time. The low runtime overhead makes this strategy suitable for highly dynamic scenarios, where the compiler micro-thread has to modify the execution flow more often.

To trigger an optimization, compiler micro-threads must observe the state of the associated computational micro-thread. If an optimization was postponed at runtime because the value of a variable was unknown at compile-time, the observed state will surely include that variable as one of the elements to be considered to decide when and how to apply the optimization at runtime, as discussed in Section 5.1.2.

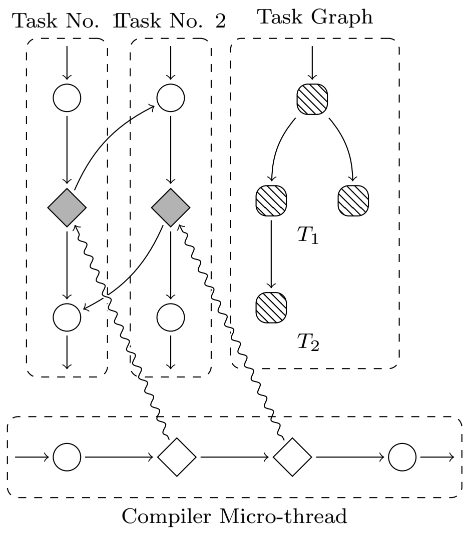
It is worth noting that our approach enables a wide range of runtime optimizations. We use branch tables to restructure the execution flow and, where this is not sufficient, we also allow code morphing to apply deeper modifications. The use of branch tables should not be perceived as just a static branch prediction [12], since it is not performed statically, but is dynamically changed every time it is needed, as a result of modifications in the execution environment.

The strong relationship between computational and compiler micro-threads motivates us to emphasize the importance of having an effective and efficient pre-scheduler. Data related to a computational micro-thread must be collected and analyzed by the corresponding compiler micro-threads. Moreover, compiler micro-threads change the behaviour of the computational micro-thread. By scheduling the different micro-threads together we aim at avoiding communication delays between them. This guarantees deterministic interactions between micro-threads, as well as high performance. Even if it is strongly discouraged, our proposal does not prevent scheduling compiler micro-threads independently. However, in this case it is required to consider explicit synchronization between micro-threads, possibly exploiting weak memory consistency models [1] to limit communication overhead.

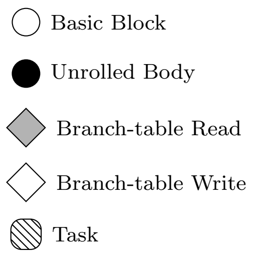
The authors of [87] observe that current production-quality compilers have issues with vectorization because the required analyses, such as in-



(a) adaptive loop unrolling



(b) dynamic task fusion



(c) legend

Figure 5.2: graphical representation of two foreseen applications of our proposed approach

terprocedural alias analysis, are not available. Such an analysis is really hard to implement at compile time, but pointers can be disambiguated at runtime. This further supports the need for splitting the compilation effort between compile-time and runtime, as allowed by our approach.

5.1.4 Foreseen Applications

In this section we present two examples of optimization that would benefit from our approach. Figure 5.2 gives a brief overview.

Adaptive Loop Unrolling

The classic loop unrolling optimizations [11] can lead to improved, unaffected or worsened execution times depending on whether the right unroll factor is chosen [20, 27]. This is a challenging task, requiring good knowledge of the target architecture [109]. In most cases, this is only available at runtime, and is exploited using a JIT compiler. Unfortunately, JITs are really heavyweight and time consuming.

With our approach, we estimate a maximum sensible unrolling factor k at compile-time. We unroll the code of the loop k times and insert a branch table read between each pair of unrolled loop bodies, as in Figure 5.2(a). This is the optimizable region. At runtime, the compiler micro-thread determines the best unrolling factor $n \leq k$ (according to the size of caches, the number of required iterations, etc.) and modifies the n-th branch table read so that it jumps back to the loop header, and all the other ones so that they either jump to the next instruction, or are substituted by `nop` instructions.

This approach is much lighter that a full-fledged JIT, but it does not enable the application of further optimizations on the unrolled code. However, if the underlying architecture is micro-programmed, the machine code will be rewritten and optimized by the hardware, making our code comparable to that unrolled by a JIT.

Dynamic Task Fusion

Task based data-flow programming models have been proven to be an attractive way to tackle some parallel applications [110]: tasks are generated on the fly, thus they require the use of a runtime scheduler to select and start them according to data and control dependencies. Therefore, after each task finishes executing, control has to return to the scheduler so that it can start the next task.

Using our approach, we can define an optimizable region just before the end of the machine code of each task, made of just a branch table read. As shown in Figure 5.2(b), at runtime, a compiler micro-thread supports the scheduler: it observes the state of the system and modifies the corresponding branch table to have it point to the beginning of the code of the next ready task. Therefore, tasks can be executed continuously, without the overhead of reaching back to the scheduler at the end of each of them. The modification of the branch table takes place as soon as the compiler micro-thread is aware of the next ready task, therefore the current and the next task will be executed one immediately after the other, as if fused together. Some call to the scheduler will still be needed, for example in order to mark a task as finished, unlocking the depending ones.

When the task graph is known at compile time, more aggressive optimizations can be performed [50]. Our approach does not allow this, but it limits the scheduling overhead that arises when inter-dependent tasks have to be executed (as tackled in [126]) and handles highly dynamic applications where the task graph is known only at runtime, even if the code is generated at compile-time.

5.1.5 Concluding Remarks and Future Work

In this section we presented a novel lightweight approach to optimize highly dynamical parallel programs, based on the use of compiler micro-threads that modify the running program at runtime, adapting it to the current environment. We described some optimizations that could be implemented using our methodology, to show it is general enough to be applied to a wide variety of algorithms. At the same time, it does not need to be completely general-purpose, since it is not meant to completely replace other techniques, such as JIT compilation.

Our methodology aims at easing the optimization of code at runtime, with low overhead. We are currently planning its implementation, to conduct an extensive and accurate experimental campaign to verify it.

The authors of [42] try to obtain similar results by generating multiple versions of code and then picking the best one at runtime. This approach, while simpler to implement, tends to generate executables that are too big, which is not an option in many environments. The lightweight compilation that we propose aims at solving this issue, by allowing an executable to change at runtime without multiple versions of the code and without a full compiler being needed to compile it just in time.

5.2 Analyzing the Sensitivity to Faults of Synchronization Primitives

[2] Modern multi-core processors provide primitives to allow parallel programs to atomically perform selected operations. Unfortunately, the increasing number of gates in such processors leads to a higher probability of faults happening during the computation. In this section, we

[2]Part of this section was previously published in [51], and it was developed in cooperation with my colleagues Ettore Speziale and Paolo Roberto Grassi, with the supervision of Professor Mariagiovanna Sami.

perform a comparison between the robustness of such primitives with respect to faults, operating at a functional level. We focus on locks, the most widespread mechanism, and on transactional memories, one of the most promising alternatives. The results come from an extensive experimental campaign based upon simulation of the considered systems. We show that locks prove to be a more robust synchronization primitive, because their vulnerable section is smaller. On the other hand, transactional memory is more likely to yield an observable wrong behaviour in the case of a fault, and this could be used to detect and correct the error. We also show that implementing locks on top of transactional memory increases its robustness, but without getting on par with that offered by locks. Finally, we present a possible fault recovery approach based on the learning techniques developed in the previous chapters.

5.2.1 Faults characterization

For our purposes, we organize faults into two different classes: the ones that affect the *general computation* - here defined *general faults* - and the ones that specifically affect mechanisms related to *critical sections*, either protected by locks or by transactions. Critical sections are particularly sensitive sections of code that are present in multi-threaded programs: wrong access to one of them by one of the concurrent threads can produce relevant errors in the program and can cause deadlocks or starvations, leading to the inability for the program to finish its execution. Hereafter we will only focus on critical section-related faults: we ignore other (general) types of faults, such as program counter corruption, that are beyond the scope of our analysis. Moreover, we aim at a technology-independent analysis: no assumptions are made here concerning the causes of faults, but we actually consider functional errors - affecting the outcome of specific instructions or operations. This will lead us to examine errors as affecting memory words or registers, often collapsing a number of different faults into one "equivalent" error type. In the same spirit, uniform random distributions will be adopted (thus abstracting from other possible distributions due to technological peculiarities).

According to Gawkowski et al. [48], the following outcomes can derive from applying faults to a program:

Correct Result (CR) the program correctly terminates its execution, computing the right value.

Incorrect Result (IR) the program gracefully terminates its execution, but the computed value is not correct and the system does not detect the error.

Fault Detected by the System (FDS) an hardware exception occurs. The system terminates the faulted program following predetermined policies.

Timeout (T) the program does not respect its timing requirements and is terminated by the system.

User-defined Message (UM) the program detects a misbehaviour, that is signalled to the user.

We follow the same classification, with the exception of User-defined Messages, since we did not add any error correction/detection machinery to the analyzed programs.

5.2.2 The Adopted Methodology

In order to obtain an indication of the respective performances of lock-based and transactional-memory-based solutions (as far as sensitivity to faults is concerned) we chose to set up an experimental environment (based on simulation tools) capable of simulating the operation of a realistic multiprocessor system as well as of supporting fault injection and simulating behavior after fault.

This choice is due to the fact that the only viable alternative would be to perform an analysis starting from the netlist of a hardware device. This device should support both lock based and transactional synchronization primitives. Moreover, it should be a neutral, publicly available benchmark (a personal choice would risk to be biased). Since such a device was not available, we decided to go for a simulation approach, so as to provide at least a first analysis that, although less precise, is more general and a good starting point for further work.

To obtain the experimental results presented here, we started from the SESC simulator. SESC is “a microprocessor architectural simulator [...] that models [...] chip multiprocessors, [...]”. CPUs used as nodes are MIPS processors, with “a full out-of-order pipeline with branch prediction, caches, buses, and every other component of a modern processor necessary for accurate simulation” [106]. More specifically, SESC operates at at functional-block level simulating the execution of a program.

In order to support the simulation of parallel programs, SESC provides its own implementation of a POSIX-like threading library, called *libapp.* libapp is much simpler than pthread, but it provides all that is needed for the aim of the present section - at least insofar as lock-based synchronization is concerned. Namely, libapp provides fork/wait primitives and lock/unlock primitives. While this allows us to proceed with the analysis of fault impact on lock-based solutions, to perform our comparison we also need an implementation of a transactional memory - which is not provided by SESC.

On the other hand, SuperTrans [103], developed by University of Florida's Advanced Computing and Information Systems Laboratory, is "a multicore, cycle-accurate and multiple issue simulator built on top of the SuperESCalar (SESC) framework that is capable of simulating three of the most common dimensions of hardware transactional memory (Eager/Eager [4, 93], Eager/Lazy [4, 104], Lazy/Lazy [54])". SuperTrans, just as SESC, is released as open source. It includes all that is part of SESC (therefore, the lock based management of memory) plus a transactional memory module. For these reasons, we chose SuperTrans as the tool for transactional-memory related simulations; being based on SESC, it granted the kind of consistency that was mandatory to compare results of simulations obtained on the two systems.

In order to explore the effects of faults, we modified SuperTrans by adding a new software component, that we named *fault injector*, allowing us to specify where and when to inject faults during the simulation, so that we can observe the outcome of the management of the mutual exclusion between two or more processes trying to access a single critical section. The fault injector can support an arbitrary number of faults. The characteristics of the faults can be completely specified by the user or randomly generated.

5.2.3 Impact of Faults on Synchronization Mechanisms

In order to evaluate how faults affect the behavior of programs run by systems that use, respectively, locks or transactional memory to protect the critical sections, we carried out an extensive experimental campaign, using a small set of synthetic benchmarks (depicted in Table 5.1) that implement well known concurrency problems, such as *shared counter* or *reader/writer interactions* [116]. Using such simple examples allows us to easily inject faults exactly in the registers and cache lines that will be accessed by the code while inside a critical section. Therefore we can verify the effect of faults on the more likely sources of problems related specifically to the synchronization mechanism adopted rather than to the general effects of faults on program's execution. Moreover, these small benchmarks share the same structure of most complex concurrent

```
Algorithm: shared_inc_lock
Data: a shared counter cnt
Result: cnt safely incremented by 1
1 lock_acquire(cnt.lock)
2 $cnt.n \leftarrow cnt.n + 1$
3 lock_release(cnt.lock)
```

Figure 5.3: Shared counter update. Locking functions guarantee *mutual exclusion* between threads while concurrently incrementing the counter

applications, so that the results we obtain are actually general. Studying the effect of faults on synchronization primitives has a direct impact on determining how the behaviour of the application will change because of them. In fact, many years of research on operating systems [116] prove the importance of the correct behaviour of such primitives.

We will now describe in detail how faults are injected in the micro-benchmarks and what are the results obtained using the *SC* micro-benchmark as a running example. Section 5.2.3 reports on fault injection in lock-based critical sections, while Section 5.2.3 refers to transactional-memory-based critical sections, Section 5.2.3 describes faulting critical sections protected with transactional-memory-based locks - a solution that, while non-realistic, allows us to complete our fault-related analysis with this alternative derived from the two basic criteria. Finally, Section 5.2.3 presents the experimental campaign setup and its results.

Lock-based Critical Sections

From the users perspective, protecting a critical section *cs* requires invoking a *lock_acquire* function before entering *cs*. This guarantees that no more than one thread at a time enters the critical section. To leave *cs*, a thread must invoke a *lock_release* function. This allows other threads to access *cs*. Figure 5.3 shows how these routines can be employed to safely increment a shared-counter.

Such locking/unlocking routines are built on top of hardware synchronization instructions, such as *atomic eXCHanGe*, *Compare And Swap*, and *Load Linked/Store Conditional.* No other *ad hoc* hardware capabilities are exploited to implement the routines: the remaining code segments are implemented using standard instructions. Figures 5.5 and 5.6 show *lock_acquire* and *lock_release* routines respectively.

Any fault generated inside a critical section can corrupt the current

```
Algorithm: $shared\_inc\_trans$
Data: a shared counter $cnt$
Result: $cnt$ safely incremented by 1
1 $trans\_begin()$
2 $cnt \leftarrow cnt + 1$
3 $trans\_commit()$
```

Figure 5.4: Shared counter update exploiting transactional memory. If a conflict is detected during a transaction, it is aborted and restarted by the hardware

```
Algorithm: $lock\_acquire$
Data: a lock $lock$
Result: $lock$ locked by current thread
1 while $XCHG(lock, LOCKED) = LOCKED$ do
2     repeat $NOP$ until $lock \neq LOCKED$
```

Figure 5.5: Implementation of *lock_acquire*

```
Algorithm: $lock\_release$
Data: a lock $lock$
Result: $lock$ unlocked
1 $lock \leftarrow UNLOCKED$
```

Figure 5.6: Implementation of *lock_release*

thread's private data, as well as the private data of other threads and shared data. This happens because a critical section's body contains only non lock-related instructions and the locking algorithm has no knowledge of the data accessed and of the instructions executed inside it.

If we consider critical section boundaries, identified by *lock_acquire* and *lock_release* routines, we see that a fault affecting data accessed by these routines is catastrophic because they control the *access to the critical section.* Even in the presence of transient faults, the program behavior is radically modified: more than one thread will access the critical section at the same time, performing a computation at the *wrong time.* The faulted program behavior matches classical concurrent programming errors, such as *lost update*, *dirty read/write*, ...

For our experimental campaign, we start by injecting faults affecting *lock_acquire*. The most important operation performed here is $XCHG$ (Figure 5.5, Line 1): it atomically replaces the memory word where *lock* resides with the $LOCKED$ constant, returning the value stored there before the swap took place. We identify three elements such that faults affecting them are critical for the synchronization process, namely: *lock*, the register containing the $LOCKED$ constant, and the return value.

To emulate faults on *lock* we consider them just by their outcome: having the program reading/writing the wrong memory location, therefore causing $XCHG$ to return a wrong value. Such value is later read (Figure 5.5, Line 1) by a comparison instruction to detect whether to enter the critical section, so this fault can allow the current thread to enter the critical section, even if the lock is not held. The program behavior cannot be predicted, and both CR and IR can be observed. A write on the wrong address could be detected, depending on the specific address, if a FDS situation (e.g. segmentation fault) occours.

Altering the $LOCKED$ word results in writing the wrong marker in the *lock* memory location. If it turns out to be equal to the valid marker $UNLOCKED$, the current thread enters the critical section without the other threads being aware that the lock has been taken. Therefore, they can in turn enter the critical section, leading to erroneous behaviour. We can observe the same behaviour also if the written marker is invalid, because every value not equal to $LOCKED$ allows the execution to enter the critical section. We can observe CR if the dynamic schedule does not result in a data race, IR or FDS otherwise.

A transient fault on the return value can result in two different behaviours: if the faulted return value is equal to $LOCKED$, the current thread spends some cycles (Figure 5.5, Lines 1 and 2) waiting for the lock to be released, without corrupting data. Otherwise, the current thread enters the critical section, incurring into a potential data race. We can observe the same program behaviour as in the previous case: CR, IR, or FDS.

As a final remark, it is worth noting that in the case of a thread trying to enter a critical section it is very unlikely to incur into T behaviour (provided only transient faults are applied, as in our experiments). For this to happen, the value accessed through the *lock* variable (Figure 5.5, Line 1) should always be equal to the value of the $LOCKED$ constant: this requires either to continuously fault *lock* in such a way to end up reading from memory locations containing the $LOCKED$ value, or to fault the return value of the $XCHG$ instruction every time in such a way that it results equal to $LOCKED$. Similar considerations apply to the spin wait loop, too (Line 2).

The *lock_release* routine is a simple store to memory. Its behaviour can be altered by injecting faults on *lock* and on the $UNLOCKED$ marker. Modifying *lock* shows the same behaviour as writing to an invalid memory address, potentially generating CR, IR, and FDS behaviours.

Finally, a fault affecting $UNLOCKED$ results in generating an invalid marker that corrupts *lock*, but the locking algorithm is not influenced: the first thread entering into the critical section restores *lock* to a consistent state. On the other hand, writing the valid but incorrect value $LOCKED$ results on T behaviour: the lock is released incorrectly, preventing any thread from entering the critical section.

Transactional Memory-based Critical Sections

In order to protect a critical section using transactional memory, the user employs three routines: *trans_begin* (instructing the transactional memory to save the current context), *trans_commit* (to publish the memory operations performed), and *trans_abort* (to explicitly terminate and restart a transaction). Figure 5.4 shows how transactional memory can be used to protect a shared counter update.

In transactional memory approach, critical section access control is distributed; every memory operation inside a critical section is validated by the transactional controller in order to detect conflicts. Detection is performed by analyzing the *read set*, (the set of memory locations read by a thread), and the *write set*, (the set of memory locations written by a thread). To emulate errors corrupting the *read set* as well as the *write set*, requires one can collapse the various fault causes into faults affecting the addresses manipulated by the transactional controller. Therefore, we will inject faults near memory access opcodes so as to affect the system immediately before memory access.

Corrupting the read set can be modeled as reading from the wrong memory location. A transactional load, LWX, both interacts with the transactional controller and fetches data from memory. As a result, the read set of the faulted processor becomes inconsistent and a wrong value is read from the memory. If the wrong value is used for subsequent computations, it can produce IRs. The same behaviour can occur even if the read value does not directly produce a corrupted value. In fact, the transactional controller could be unable to detect a conflict due to the corrupted read set, thus allowing a transaction to commit when it should have been aborted instead. Moreover, if the corrupted address is later used for a memory store to a location not accessible by the faulted processors, a FDS occurs.

If faults lead to corrupting the write set, the same behaviour can observed. In this case the faulted instruction is the transactional write, SWX; as in the case of LWX, the instruction also interacts both with the transactional controller and the memory. Depending on the fault-affected value, an IR or FDS can occur. The difference with respect to faulting the read set is that an FDS can occur immediately.

Hardware implementations of transactional memory introduce three new opcodes, namely $XBEGIN$, $XCOMMIT$, and $XABORT$ respectively implementing $trans\_begin$, $trans\_commit$, and $trans\_abort$. All these operations do not use general purpose hardware; they interact directly with the transactional memory controller, thus to simulate faults relative to them we cannot just inject faults into registers or non-transactional memory, but we have to fault the simulated hardware primitives themselves. Faults concerning this scenario corrupt processor context saved by $XBEGIN$ and restored by $XCOMMIT$ and $XABORT$. Since these faults would be very much dependent on a specific implementation and technology, we do not consider them; obviously, extending the set of faults would increase the sensitivity to faults of the system.

Transactional Locking-based Critical Sections

As shown in Section 5.2.3, we can inject a wide variety of faults on locks, but the lock is directly manipulated only at critical section bounds, so there is not much possibility for such faults to happen. Every other fault happening inside a critical section protected by locks is not related to locks themselves: as such it could happen whatever the synchronization primitive being used, and is therefore not interesting for this study. On the other hand, transactional memory is vulnerable to a narrow class

```
Algorithm: trans_xchg
Data: an address addr
Data: a value val
Result: val written into addr, old value returned
1 trans_begin()
2 old ← mem[addr]
3 mem[addr] ← val
4 trans_commit()
5 return old
```

Figure 5.7: Atomic exchange implemented using transactional memory. It is used as a building block for transactional memory-based locks

```
[upReg]
generator = 'uniform'
regNo = 'R18'
kind  = 'bitFlip'
atTime = 1100
```

Figure 5.8: An example of fault taken from the configuration file. A bit-flip fault named `upReg` will be applied to register `R18` at 1100^{th} cycle of the simulation

of faults, see Section 5.2.3, but they expose more faulting opportunity because as long as the transaction is active every memory access could be influenced by faults in the *read set* or the *write set.*

This observation led us to try to implement locks “on top of” transactional constructs. While this is not a viable solution for real systems, it allows us to study whether transactional memory helps reducing faulting opportunities. The locking and unlocking algorithms are the same used for lock-based critical sections (Figure 5.5 and 5.6). In order to exploit transactional memory, we replaced the $XCHG$ instruction with an equivalent routine written using transactional constructs. Its implementation can be seen in Figure 5.7. The lock release routine has been modified so as to be protected by a transaction.

For faults happening inside the critical section we can make the same observations as for locks, because the critical section does not contain any special instruction.

On the other hand, we note that injecting faults on critical section boundaries requires injecting faults on the transactions protecting the atomic exchange. The kind of faults that can be injected are the same as for transactional memory: basically, we can fault the read set and the write set.

In this particular critical section, the read set and the write set are identical: they consist just of the word used to store the lock. Faulting the lock address can thus produce FDS, IR, or T behaviours. The first arises when the faulted lock address refers to a memory region that cannot be written by the faulted thread. If the word identified by the faulted address can be written, a data race can occur, possibly generating either CR or IR. The T behaviour occurs when reading from the faulted address causes the faulted thread to spend too much time in the lock busy-wait loop.

Results of the Experimental Campaign

Our experimental campaign focused on the micro-benchmarks reported in Table 5.1. We coded each micro-benchmark in three different flavour, each employing a different primitive to protect its critical sections. The *lock* flavour, uses locks, *trans* uses transactions, while *trans-lock* adopts locks implemented by means of transactions.

Each micro-benchmark was first run without applying any faults. Observing the execution trace we detected points where faults could be injected, as suggested in Section 5.2.3, 5.2.3, and 5.2.3. Each flavour exposes different faultable points. Faults will affect execution with the *lock* flavour while acquiring and releasing the lock. The *trans* flavour is faultable while accessing the read set and the write set, i.e. near each LWX and SWX. The *trans-lock* flavour exposes the same faultable points as *trans*, but the critical section is shorter.

We wanted to see the evolution of the behaviour of the micro-benchmarks subject to an increasing number of faults. Therefore, for each of them we injected an increasing number of faults, from 1 to 4. For each benchmark, for each given number of faults, we performed 960 [3] runs. Before each run we randomly extract i faultable points taken from those observed by analyzing the execution trace. To allow for some randomness, each fault was randomly applied between 1 and 4 cycles before the time instant it was registered in the original execution trace. In case of faults applied to registers, we randomly generated the number of the register bit to fault. For faults applied to cache line reads, we randomly generated the loaded word bit to fault. Figure 5.8 shows a generated fault entry in the SESC configuration file format.

[3] 240 for RWL-trans.

Table 5.2 reports individual benchmark results, while Figure 5.9 shows the percentage of *CR*, *IR*, *FDS*, and *T* for each flavour, varying the number of applied faults.

The *lock* flavour is the most robust, because there are fewer points where a fault can be injected. Moreover, the fault must be injected at a precise time, otherwise the locking algorithm tends to mask the fault and thus overcomes a previous soft fault. In fact, the locking algorithm usually rewrites the content of the lock word at the beginning of the critical section, while trying to acquire the ownership via the $XCHG$ instruction, and at its end, while releasing the lock. Moreover, not all faults injected on locks can be observed, because even if two threads happen to enter in a critical section at the same time, they could not incur in a data race, depending on the specific scheduling taking place.

Looking at Table 5.1 we see that the *trans* flavour obtains the worst outcome, with less CR compared to the *lock* flavour, because transactional memory exposes more faultable points. However, the probability that a fault will be detected (FDS) is greater, because most failures are due to accesses to wrong memory areas. These are detected by the operating system and could, in principle, be used to perform error correction, thus increasing the number of correct results.

Finally, implementing locks on top of transactional memory, i.e. the *trans-lock* flavour, increases the robustness with respect to *trans*, because each transaction lasts only as long as needed to change the lock value, but it cannot achieve the robustness of the *lock* flavour, because as short as that time span can be, every single access to memory during it can be subject to faults. Let us now analyse in detail the outcome of each benchmark.

Shared Counter and Shared Multi Counter the critical section associated to *SC* is the shorter of all the benchmark suite, while *SMC* employs a longer critical section, updating more than a shared counter at time.

The *lock* flavour is the most susceptible to short critical sections. Indeed, on such scenario the program hot spot is *lock acquisition*, so any fault that induces spending some extra cycles in the lock waiting loop, greatly lowers performance, generating a considerable amount of T behaviour. When increasing the length of the critical section, the number of T behavior decreases, as shown by the *SMC* micro-benchmark, where we can observe a greater number of CR.

Both *trans* and *trans-lock* flavours follow the same trends in both *SC* and *SMC*. To obtain T behaviour, read and/or write sets of a transaction must be faulted in order to read/write data from/to the shared data,

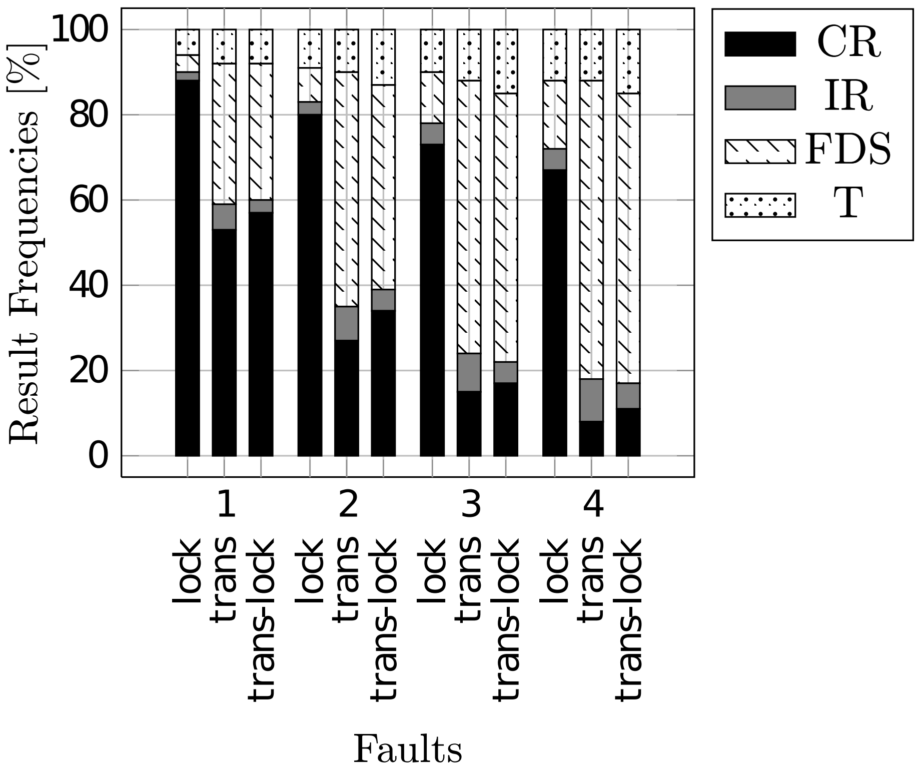


Figure 5.9: Distribution of benchmark results, varying the number of applied faults

Table 5.1: Benchmarks

SC	concurrent increment of a shared counter. Each thread performs 8 atomic increments.
SMC	concurrent increment of shared counters. Each thread executes 4 critical sections, incrementing 16 counters each time.
RW	reader/writer problem. Threads are partitioned into two equally sized sets: readers and writers. Writers produce items writing them into a global buffer. Readers read items from the buffer. When all items have been produced, the readers concurrently write all the read items into another buffer read by the main thread to perform a final sanity check. Buffers are implemented using arrays.
RWL	reader/writer problem. Same behaviour of *RW*, but shared buffers are implemented using single-linked lists.

Table 5.2: Benchmark results. For each configuration, 960 runs have been performed (240 for RWL-trans)

Benchmark		**CR** [%]				**IR** [%]				**FDS** [%]				**T** [%]			
		$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$
SC	lock	82	76	71	63	3	5	6	9	4	9	13	17	11	10	10	11
	trans	50	25	13	7	8	12	14	14	42	63	73	79	0	0	0	0
	trans-lock	55	35	18	10	5	6	8	8	40	59	74	82	0	0	0	0
SMC	lock	92	86	81	80	4	4	6	5	3	8	9	11	1	2	4	4
	trans	51	24	14	10	8	10	12	14	40	64	73	74	1	2	1	2
	trans-lock	57	34	21	13	6	10	10	11	37	55	69	75	0	1	0	1
RW	lock	87	77	68	61	3	3	4	5	4	9	15	19	6	11	13	15
	trans	59	33	18	9	3	4	4	5	19	38	46	56	19	25	32	30
	trans-lock	58	30	12	11	1	2	7	2	25	38	54	59	16	30	27	28
RWL	lock	89	83	73	64	0	1	2	3	4	8	10	16	7	8	15	17
	trans	48	19	11	4	3	2	3	3	27	48	60	64	22	31	26	29
	trans-lock	56	35	17	10	1	2	2	5	26	39	54	58	17	24	27	27

forcing an abort. Both the micro-benchmarks have a small amount of shared data, so the probability of this outcome is negligible.

Reader/Writer and Reader/Writer List the *RW* micro-benchmark uses arrays to implement shared buffers, while *RWL* relies on single-linked list, thus critical sections are longer and access memory more frequently.

Locking-based techniques exhibit the same behaviour in the two micro-benchmarks. On the other hand, the *trans* flavour is heavily influenced by using single-linked lists. Using more complex structures results in more memory accesses, mostly related to list navigation. Thus, the probability of incurring into a *FDS* increases.

5.2.4 Fault recovery guided by machine learning

This section presented an analysis of the sensitivity to faults of locks and transactional memories.

In this subsection we will show how we can to recover from hardware faults affecting the proper execution of programs by using an approach based on long-term learning, similar to the one presented in Chapter 3.

First of all, we need to detect that a fault has happened. This can be done with either a lock-based or a transactional memory based synchronization system.

As we previously stated, there are three possible execution outcomes indicating a failure: Incorrect Result (IR), Fault Detected by the System (FDS) or Timeout (T).

We cannot detect IRs because that would require having an oracle able to predict the correct result of the execution. This was possible in the case of the tests we run for this fault sensitivity analisys, but is not applicable in a realistic environment. Though, as shown in Figure 5.9, IRs are just a small fraction of the failing executions, so even by just focusing on the other ones we can greatly improve the behaviour of the system.

On the other hand, FDS and T faults are easily detected in a real environment using the same technique used for this analysis. When we know a program has failed because of a fault, we can try to apply various recovery algorithms. Here we present two of them, based on long-term learning.

Compilation-based approach

This approach is a direct application of the one described in Chapter 3.

Let us suppose that we have a compiler that is able to statically define, at compile time, what system resources to assign to a program, basing the decision upon a set of heuristics automatically learned by the long-term learning algorithm.

When a fault is detected, the program is recompiled from scratch. During the compilation process, different candidate versions of the program will be tested to determine their performance.

The candidates trying to use faulted components will fail to execute correctly, therefore they will be automatically penalized by the scoring system of long-term learning, thus leading to the creation of heuristics able to only compile programs using the non-faulted resources of the system.

The implementation would be particolarly lightweight: the learning of the heuristics should be activated only during the recovery after the failure of a program. For every other compilation, we could just use the currently existing ones, knowing that they are already able to perform a correct allocation of the resources.

Runtime-based approach

The previous approach only involves the compiler, so it is general and can be applied to every program, without limitations.

Nevertheless, it can be useful, in some cases, to be able to not recompile a program to deal with a fault. For example, in case of a server providing a high availability service, it could be problematic to shut down the program to substitute it with a new executable.

Therefore, a runtime library could be implemented, responsible for managing the allocation of the resources requested by the program. This library would perform the allocation choices using a set of heuristics learned through long-term learning. Every time a decision has to be made, the heuristics are used and evolved. Therefore, at the time of the evaluation of the heuristics there is always complete information about the current status of the system, and the heuristics are actually able to guide the execution in such a way to avoid using the faulted components.

This runtime-based approach requires the long-term learning algorithm to be implemented inside the runtime library itself.

5.2.5 Concluding Remarks

In this section we analyzed the behavior of locks and transactional memory when they are affected by faults. We injected from 1 up to 4 faults during the execution of selected benchmarks and analyzed the outcome of the execution. As it is easy to understand, while the number of faults grows, the probability of a visible failure increases. The important result is that locks proved to be more fault resilient because they expose a smaller “faultable surface”, and it is therefore more unlikely for a fault to have the execution fail. We did not consider a specific hardware implementation, and focused only on observing the functionality of the synchronization primitives under the effect of faults. Nonetheless, it should be considered that transactional memory requires specialized hardware components to be added to the system, and this components are themselves subject to faults. This suggests that the actual fault tolerance of transactional memory could be lower than our results suggest. Further experimental campaign should be conducted in order to prove this point. On the other hand, our results show that with transactional memory the system is more likely to be able to detect the presence of faults. Further experiments could determine whether fault detection and recovery capabilities could be more effective or easier to implement in a transactional memory based system. Finally, we presented a possible fault recovery system based on the machine learning techniques detailed in the previous chapters.

6 Conclusion

The complexity of modern computing architecture makes the task of fully exploiting them increasingly hard. Heterogeneity and the growing level of parallelism, with multi-core processors nowadays being commonplace even in smartphones and embedded systems and with many-cores becoming more and more widespread, poses new challenges to programmers and, especially, compiler writers.

In this dissertation I presented my contributions towards the goal of enabling programs to better adapt to the characteristics of the available underlying hardware. Most of them are related to machine learning techniques applied to the compilation process.

In particular, the first and biggest contribution is the definition of a novel evolutionary algorithm, called *Long-term learning*. In order to take decisions about the optimization algorithms that adapt the programs to the target architecture, compilers have to use heuristics because they cannot precisely predict the outcome of applying them. Traditionally, such heuristics have been hand-written by compiler experts, but this is a time consuming and error prone task.

Long-term learning is meant to automatically build good compilation heuristics for a compiler that has none, adapting automatically to the target architecture, exploting its characteristics and making the most out of the available optimization algorithms. At the same time, it aims at building readable heuristics instead of just a model that is working but hard to understand.

We showed that long-term learning is able to efficiently perform this task by gathering experimental results on three different hardware configurations, using two different implementations of our algorithm on two compilers, GCC and PetaBricks. The results show that, in a short time, long-term learning is able to find readable heuristics that are as good as the handwritten ones an expert could come up with. Given enough time, it can further improve over them, surpassing the maximum level of performance they can reach.

The main novelty of long-term learning with respect to other algorithms having a similar objective consists in generating, heuristics based on human-readable mathematical formulas instead of some statistical model much harder to understand. Furthermore, it also learns sets of heuristics considering them as a whole, therefore taking their interactions into account, instead of assuming they are independent. Moreover, long-term learning has no need for an inital training phase, that is usually required by other machine learning based compilation approaches. It is an online learning algorithm and it keeps learning every time a new program is compiled, acquiring knowledge and using it for the next compilations.

Long-term learning is partially based upon iterative compilation. As such, it needs to test multiple candidate versions of the program it is compiling. Because of this, this thesis presented a method to parallelize the testing of candidates of a generic iterative compilation approach using MapReduce. The parallelization can be over a set of identical machines in a cluster or, under certain conditions, on a single machine, partitioning its computational resources. The efficacy and efficiency of this method were proven by implementing it into the PetaBricks compiler and by conduction an extensive experimental campaing showing that it can reach almost linear speedups over four different hardware configurations. Still, the method is general and can be applied to most iterative compilation approaches, allowing them to better exploit the parallelism provided by many modern computing architectures.

Finally, a couple of possible future works are suggested as well. First, to complete the contribution to the development of compilers targeted at modern architectures, I presented a proposal for a new lightweight approach to compiling parallel programs. This approach, jointly developed with a colleague of mine, divides the burden of applying compiler optimizations between compile time and runtime. As much work as possible is done at compile time, preventing the need to have a full-fledged compiler at runtime. The choice of what optimizations to postpone at runtime is analogous to those tipically faced by iterative compilation and machine learning techniques applied to compilation. Therefore, this thesis presented a method to apply a technique derived from long-term learning in order to define the heuristics that will take care of making such decision.

Second, starting from a project, developed with some colleagues, about an analysis of the sensitivity to hardware faults of various synchronization primitives, a novel approach for fault recovery is suggested, using heuristics obtained through long-term learning to determine the allocation of the required system resources while avoiding the faulted components.

Publications

During my PhD studies I explored various research topics. This thesis derives from the main one I chose and developed more thoroughly. Still, some other works were complete enough to be published or presented at various international or national conferences and venues.

Here is a list of all of them.

- M. Tartara, S. Crespi Reghizzi. *Continuous Learning of Compiler Heuristics.* To appear in the ACM Transactions on Architecture and Code Optimization, and to be presented at the 8th International Conference on High-Performance and Embedded Architectures and Compilers, January 21-23, 2013, Berlin, Germany.

- M. Tartara, S. Crespi Reghizzi. *Parallel Iterative Compilation: Using MapReduce to Speedup Machine Learning in Compilers.* The Third International Workshop on MapReduce and its Applications (MAPREDUCE'12), HPDC'2012, Delft, the Netherlands, June 18-19, 2012.

- E. Speziale, M. Tartara. *A Lightweight Approach to Compiling and Scheduling Highly Dynamic Parallel Programs.* Poster session of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar'12), Berkeley CA, USA, June 7-8, 2012.

- P. R. Grassi, M. Sami, E. Speziale, M. Tartara. *Analyzing the Sensitivity to Faults of Synchronization Primitives*, IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'11), Vancouver, Canada, October 3-5, 2011.

- M. Tartara, S. Crespi Reghizzi and S. Campanoni. *Extending hammocks for parallelism detection.* Italian Conference on Theoretical Computer Science (ICTCS). Camerino, Italy, 15-17 September 2010.

- C. Silvano, W. Fornaciari, S. Crespi Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, A. Di Biagio, E.

Speziale, M. Tartara, D. Siorpaes, H. Huebert, B. Stabernack, J. Brandenburg, M. Palkovic, P. Raghavan, C. Ykman-Couvreur, A. Bartzas, S. Xydis, D. Soudris, T. Kempf, G. Ascheid, R. Leupers H. Meyr, J. Ansari, P. Mahonen, and B. Vanthournout, *2PARMA: Parallel Paradigms and Run-time Management Techniques for Many-core Architectures*, ISVLSI 2010: IEEE Annual Symposium on VLSI, pages 494-499, Lixouri, Kefalonia - Greece, July 2010.

- S. Campanoni, M. Tartara, S. Crespi Reghizzi, *ILDJIT: A parallel, free software and highly flexible Dynamic Compiler.* Conferenza Italiana sul Software Libero, Cagliari, June 2010.

- M. Tartara, S. Campanoni, G. Agosta and S. Crespi Reghizzi. *Parallelism and Retargetability in the ILDJIT Dynamic Compiler.* in ARCS '10 - 23th International Conference on Architecture of Computing Systens 2010 - Workshop Proceedings, Hannover, February 2010, pp. 285-291.

- M. Tartara, S. Campanoni, G. Agosta, and S. Crespi Reghizzi. *Just-in-time compilation on ARM processors.* In ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, pages 70–73, New York, NY, USA, 2009. ACM.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Felix V. Agakov, Edwin V. Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *CGO*, pages 295–305. IEEE Computer Society, 2006.

[3] Lelac Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding Effective Compilation Sequences. In David B. Whalley and Ron Cytron, editors, *LCTES*, pages 231–239. ACM, 2004.

[4] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327. IEEE Computer Society, 2005.

[5] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. PetaBricks: a Language and Compiler for Algorithmic Choice. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 38–49. ACM, 2009.

[6] Jason Ansel, Maciej Pacula, Saman P. Amarasinghe, and Una-May O'Reilly. An Efficient Evolutionary Algorithm for Solving Incrementally Structured Problems. In Natalio Krasnogor and Pier Luca Lanzi, editors, *GECCO*, pages 1699–1706. ACM, 2011.

[7] Claudio Arlandini and Alice Invernizzi. LAGRANGE: un nuovo server per il calcolo ad alte prestazioni. *Bollettino del CILEA*, 0(110), 2008.

[8] Matej Artac, Matiaz Jogan, and Ales Leonardis. Incremental PCA for On-line Visual Learning and Recognition. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, pages 781 – 784 vol.3, 2002.

[9] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker Lester, John Shalf, Samulel W. Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[10] John Aycock. A Brief History of Just-In-Time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[11] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[12] Thomas Ball and James R. Larus. Branch Prediction For Free. In *PLDI*, pages 300–313, 1993.

[13] Michel Barreteau, François Bodin, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John R. Gurd, Jan Hoogerbrugge, Ping Hu, William Jalby, Toru Kisuki, Peter M. W. Knijnenburg, Paul van der Mark, Andy Nisbet, Michael F. P. O'Boyle, Erven Rohou, André Seznec, Elena Stöhr, Menno Treffers, and Harry A. G. Wijshoff. OCEANS - Optimising Compilers for Embedded Applications. In Patrick Amestoy, Philippe Berger, Michel J. Daydé, Iain S. Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par*, volume 1685 of *Lecture Notes in Computer Science*, pages 1171–1175. Springer, 1999.

[14] Avrim Blum. On-line Algorithms in Machine Learning. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 306–325. Springer, 1996.

[15] François Bodin, Toru Kisuki, Peter M.W. Knijnenburg, Micheal F.P. O'Boyle, and Erven Rohou. Iterative Compilation in a Non-Linear Optimisation Space, 1998.

[16] Shekhar Borkar. Thousand Core Chips – A Technology Perspective. In *DAC*, pages 746–749, 2007.

[17] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas B. Jablin, and David I. August. Revisiting the Sequential Programming Model for the Multicore Era. *IEEE Micro*, 28(1):12–20, 2008.

[18] Simone Campanoni, Giovanni Agosta, Stefano Crespi Reghizzi, and Andrea Di Biagio. A highly flexible, parallel virtual machine: Design and experience of ildjit. *Softw., Pract. Exper.*, 40(2):177–207, 2010.

[19] Simone Campanoni, Martino Sykora, Giovanni Agosta, and Stefano Crespi Reghizzi. Dynamic Look Ahead Compilation: A Technique to Hide JIT Compilation Latencies in Multicore Environment. In Oege de Moor and Michael I. Schwartzbach, editors, *CC*, volume 5501 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2009.

[20] Steve Carr and Ken Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1768–1810, 1994.

[21] John Cavazos, Grigori Fursin, Felix V. Agakov, Edwin V. Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *CGO*, pages 185–197. IEEE Computer Society, 2007.

[22] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In Y. Annie Liu and Reinhard Wilhelm, editors, *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9. ACM, 1999.

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

[24] Issam Dagher. Incremental PCA-LDA algorithm. In *Computational Intelligence for Measurement Systems and Applications (CIMSA), 2010 IEEE International Conference on*, pages 97 –101, sept. 2010.

[25] R. J. Dakin and Peter C. Poole. A Mixed Code Approach. *Comput. J.*, 16(3):219–222, 1973.

[26] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David A. Patterson, John Shalf, and Katherine A. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures. In *SC*, page 4. IEEE/ACM, 2008.

[27] Jack W. Davidson and Sanjay Jinturkar. Aggressive Loop Unrolling in a Retargetable Optimizing Compiler. In *CC*, pages 59–73, 1996.

[28] J. L. Dawson. Combining Interpretive Code with Machine Code. *Comput. J.*, 16(3):216–219, 1973.

[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[30] Pedro C. Diniz and Martin C. Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. In Marina C. Chen, Ron K. Cytron, and A. Michael Berman, editors, *PLDI*, pages 71–84. ACM, 1997.

[31] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[32] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

[33] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, and Olivier Temam. Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction. In Utpal Banerjee, José Moreira, Michel Dubois, and Per Stenström, editors, *Conf. Computing Frontiers*, pages 131–142. ACM, 2007.

[34] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez, editors, *MICRO*, pages 78–88. ACM, 2009.

[35] Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009.

[36] Michael Fairley. mincemeat.py - MapReduce on Python. http://remembersaurus.com/mincemeatpy/, 2012 (retrieved).

[37] David B. Fogel. What is evolutionary computation? *Spectrum, IEEE*, 37(2):26, 28 –32, feb 2000.

[38] Björn Franke, Michael F. P. O'Boyle, John Thomson, and Grigori Fursin. Probabilistic source-level optimisation of embedded programs. In Yunheung Paek and Rajiv Gupta, editors, *LCTES*, pages 78–86. ACM, 2005.

[39] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[40] Grigori Fursin. Collective benchmark (cBench), A collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization., 2010. http://ctuning.org/cbench.

[41] Grigori Fursin, John Cavazos, Michael F. P. O'Boyle, and Olivier Temam. MiDataSets: Creating the Conditions for a More Realistic Evaluation of Iterative Optimization. In Koen De Bosschere, David R. Kaeli, Per Stenström, David B. Whalley, and Theo Ungerer, editors, *HiPEAC*, volume 4367 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2007.

[42] Grigori Fursin, Albert Cohen, Michael F. P. O'Boyle, and Olivier Temam. A Practical Method for Quickly Evaluating Program Optimizations. In Thomas M. Conte, Nacho Navarro, Wen mei W. Hwu, Mateo Valero, and Theo Ungerer, editors, *HiPEAC*, volume 3793 of *Lecture Notes in Computer Science*, pages 29–46. Springer, 2005.

[43] Grigori Fursin, Albert Cohen, Michael F. P. O'Boyle, and Olivier Temam. Quick and Practical Run-Time Evaluation of Multiple Program Optimizations. *T. HiPEAC*, 1:34–53, 2007.

[44] Grigori Fursin, Cupertino, Miranda, Sebastian Pop, Albert Cohen, and Olivier Temam. Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation. In *GCC Developpers Summit*, 2007.

[45] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson,

Christopher K. I. Williams, and Michael F. P. O'Boyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[46] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. Milepost GCC: Machine Learning Based Research Compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

[47] Grigori Fursin and Olivier Temam. Collective Optimization: a Practical Collaborative Approach. *TACO*, 7(4):20, 2010.

[48] Piotr Gawkowski, Janusz Sosnowski, and B. Radko. Analyzing the Effectiveness of Fault Hardening Procedures. In *IOLTS*, pages 14–19. IEEE Computer Society, 2005.

[49] Dan Gillick, Arlo Faria, and John Denero. MapReduce: Distributed Computing for Machine Learning, 2006.

[50] Michael I. Gordon, William Thies, and Saman P. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ASPLOS*, pages 151–162, 2006.

[51] Paolo Roberto Grassi, Mariagiovanna Sami, Ettore Speziale, and Michele Tartara. Analyzing the Sensitivity to Faults of Synchronization Primitives. In *DFT*, pages 349–355. IEEE, 2011.

[52] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: a Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[53] Andrei Hagiescu, Huynh Phung Huynh, Weng-Fai Wong, and Rick Siow Mong Goh. Automated Architecture-Aware Mapping of Streaming Applications Onto GPUs. In *IPDPS*, pages 467–478, 2011.

[54] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113. IEEE Computer Society, 2004.

[55] Gilbert Josep Hansen. *Adaptive Systems for the Dynamic Runtime Optimization of Programs.* PhD thesis, 1974.

[56] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[57] Urs Hölzle and David Ungar. A Third-Generation SELF Implementation: Reconsiling Responsiveness with Performance. In *OOPSLA*, pages 229–243, 1994.

[58] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In Mary Lou Soffa and Evelyn Duesterwald, editors, *CGO*, pages 165–174. ACM, 2008.

[59] John H. Howard. On Overview of the Andrew File System. In *USENIX Winter*, pages 23–26, 1988.

[60] Michael R. Jantz and Prasad A. Kulkarni. Eliminating false phase interactions to reduce optimization phase order search space. In Kathail et al. [64], pages 187–196.

[61] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *CoRR*, cs.AI/9605103, 1996.

[62] Sam Kamin, Baris Aktemur, and Michael Katelman. Staging static analyses for program generation. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE*, pages 1–10. ACM, 2006.

[63] Ralf Karrenberg and Sebastian Hack. Whole-function Vectorization. In *CGO*, pages 141–150, 2011.

[64] Vinod Kathail, Reid Tatge, and Rajeev Barua, editors. *Proceedings of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2010, Scottsdale, AZ, USA, October 24-29, 2010.* ACM, 2010.

[65] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*, 2010.

[66] Youfeng Wu Kingsum Chow. Feedback-Directed Selection and Characterization of Compiler Optimizations. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*, 1999.

[67] Toru Kisuki, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, François Bodin, and Harry A. G. Wijshoff. A Feasibility Study in Iterative Compilation. In Constantine D. Polychronopoulos, Kazuki Joe, Akira Fukuda, and Shinji Tomita, editors, *ISHPC*, volume 1615 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 1999.

[68] Toru Kisuki, Peter M.W. Knijnenburg, Micheal F.P. O'Boyle, and Harry A. G. Wijshoff. Iterative Compilation in Program Optimization, 2000.

[69] Peter M. W. Knijnenburg, Toru Kisuki, and Michael F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.

[70] Donald E. Knuth. Backus normal form vs. Backus Naur form. *Commun. ACM*, 7(12):735–736, December 1964.

[71] Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[72] Ron Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*, pages 1137–1145. Morgan Kaufmann, 1995.

[73] Flip Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 201–212. ACM, 2000.

[74] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot^{TM}Client Compiler for Java 6. *TACO*, 5(1), 2008.

[75] John Koza and Riccardo Poli. Genetic Programming. In Edmund K. Burke and Graham Kendall, editors, *Search Methodologies*, pages 127–164. Springer US, 2005.

[76] Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the Overhead of Dynamic Compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.

[77] Prasad A. Kulkarni, Stephen Hines, Jason Hiser, David B. Whalley, Jack W. Davidson, and Douglas L. Jones. Fast searches for effective optimization phase sequences. In William Pugh and Craig Chambers, editors, *PLDI*, pages 171–182. ACM, 2004.

[78] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[79] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: using hot optimizations without getting burned. In Michael I. Schwartzbach and Thomas Ball, editors, *PLDI*, pages 239–251. ACM, 2006.

[80] Hugh Leather. *Machine Learning in Compilers*. PhD thesis, Institute of Computing Systems Architecture, School of Informatics, University of Edinburgh, 2011.

[81] Hugh Leather, Edwin V. Bonilla, and Michael F. P. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *CGO*, pages 81–91. IEEE Computer Society, 2009.

[82] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jung-Ho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. An OpenCL Framework for Heterogeneous Multicores with Local Memory. In *PACT*, pages 193–204, 2010.

[83] Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. An OpenCL Framework for Homogeneous Manycores with No Hardware Cache Coherence. In *PACT*, pages 56–67, 2011.

[84] Shun Long. Sustainable Learning-Based Optimization Based on RKNN Outlier Detection. In *5th Workshop on Statistical and Machine learning approaches to Architecture and Compilation (SMART 2011)*, 2011.

[85] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[86] Zhiqiang Ma and Lin Gu. The limitation of mapreduce: A probing case and a lightweight solution. In *CLOUD COMPUTING 2010: Proc. of the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization*, pages 68–73, 2010.

[87] Saeed Maleki, Yaoqing Gao, María Jesús Garzarán, Tommy Wong, and David A. Padua. An Evaluation of Vectorizing Compilers. In *PACT*, pages 372–382, 2011.

[88] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: online contention detection and response. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *CGO*, pages 257–265. ACM, 2010.

[89] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*. Red Hat, Inc., 2003.

[90] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In Donia Scott, editor, *AIMSA*, volume 2443 of *Lecture Notes in Computer Science*, pages 41–50. Springer, 2002.

[91] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), 1965.

[92] Gordon E. Moore. Excerpts from A Conversation with Gordon Moore: Moore's Law, 2005.

[93] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265. IEEE Computer Society, 2006.

[94] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. Practical aggregation of semantical program properties for machine learning based optimization. In Kathail et al. [64], pages 197–206.

[95] Hugo Viera. Neto and Nehmzow Ulrich. Incremental PCA: An alternative approach for novelty detection. In *Proc. Towards Autonomous Robotic Systems (TAROS'05)*, 2005.

[96] Andy Nisbet. GAPS: A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation. In Peter M. A. Sloot, Marian Bubak, and Louis O. Hertzberger, editors, *HPCN Europe*, volume 1401 of *Lecture Notes in Computer Science*, pages 987–989. Springer, 1998.

[97] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *CGO*, pages 151–160, 2011.

[98] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, 2009.

[99] OpenMP Architecture Review Board. *OpenMP Application Program Interface, version 3.0*, 2008.

[100] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO*, pages 105–118. IEEE Computer Society, 2005.

[101] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In Rajesh K. Gupta and Vincent John Mooney, editors, *CASES*, pages 65–74. ACM, 2011.

[102] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[103] James Poe, Chang-Burm Cho, and Tao Li. Using Analytical Models to Efficiently Explore Hardware Transactional Memory and Multi-Core Co-Design. In *SBAC-PAD*, pages 159–166. IEEE Computer Society, 2008.

[104] Ravi Rajwar, Maurice Herlihy, and Konrad K. Lai. Virtualizing Transactional Memory. In *ISCA*, pages 494–505. IEEE Computer Society, 2005.

[105] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*, pages 13–24. IEEE Computer Society, 2007.

[106] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, 2011.

[107] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.).* Pearson Education, 2010.

[108] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C.

Fogarty, editors, *EuroGP*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 1998.

[109] Vivek Sarkar. Optimized Unrolling of Nested Loops. In *ICS*, pages 153–166, 2000.

[110] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-memory Multicore Systems. In *SC*, 2009.

[111] E. Speziale and M. Tartara. A Lightweight Approach to Compiling and Scheduling Highly Dynamic Parallel Programs. 2012.

[112] Mark Stephenson and Saman P. Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *CGO*, pages 123–134. IEEE Computer Society, 2005.

[113] Mark Stephenson, Saman P. Amarasinghe, Martin C. Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In Ron Cytron and Rajiv Gupta, editors, *PLDI*, pages 77–90. ACM, 2003.

[114] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *CGO*, pages 111–119, 2010.

[115] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[116] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation.* Prentice Hall, third edition, 2006.

[117] Michele Tartara, Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. Just-In-Time compilation on ARM processors. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 70–73, New York, NY, USA, 2009. ACM.

[118] Michele Tartara, Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. Parallelism and Retargetability in the ILDJIT Dynamic Compiler. In Michael Beigl and Francisco J. Cazorla-Almeida, editors, *ARCS Workshops*, pages 285–291. VDE Verlag, 2010.

[119] Michele Tartara and Stefano Crespi Reghizzi. Parallel Iterative Compilation: Using MapReduce to Speedup Machine Learning in Compilers. In *Proceedings of third international workshop on MapReduce and its Applications Date*, MapReduce '12, pages 33–40, New York, NY, USA, 2012. ACM.

[120] The Apache Software Foundation. Hadoop MapReduce. `http://hadoop.apache.org/mapreduce`, Feb 2012 (retrieved).

[121] John Thomson, Michael F. P. O'Boyle, Grigori Fursin, and Björn Franke. Reducing training time in a one-shot machine learning-based compiler. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, *LCPC*, volume 5898 of *Lecture Notes in Computer Science*, pages 399–407. Springer, 2009.

[122] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary W. Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS*, pages 1–12. IEEE, 2009.

[123] Ananta Tiwari, Jeffrey K. Hollingsworth, Chun Chen, Mary W. Hall, Chunhua Liao, Daniel J. Quinlan, and Jacqueline Chame. Auto-tuning full applications: A case study. *IJHPCA*, 25(3):286–294, 2011.

[124] Robert M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1), 1967.

[125] Guido van Rossum. Python Programming Language. In *USENIX Annual Technical Conference*. USENIX, 2007.

[126] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. A Unified Scheduler for Recursive and Task Dataflow Parallelism. In *PACT*, pages 1–11, 2011.

[127] Michael Voss and Rudolf Eigenmann. Adapt: Automated decoupled adaptive program transformation. In *ICPP*, pages 163–, 2000.

[128] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[129] R. Clinton Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC*, page 38. IEEE, 1998.

[130] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1–3):37 – 52, 1987. Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists.

[131] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. In *MICRO*, pages 51–61, 1991.

[132] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.